18 Apr **2013**

Quick and Useful Tricks for Analyzing Binaries for Pen Testers

0 comments Posted by eskoudis
Filed under File Analysis, Methodology

[Editor's Note: In the article below, Yori Kvitchko kicks off a series of brief posts about quick and dirty but very useful techniques pen testers can apply to analyze stand-alone files (such as binaries, Flash files, etc.) that they encounter in penetration tests. There is a treasure trove of info in most stand-alone files you'll encounter, and the ability to quickly go through those files to pry out their secrets is hugely helpful in penetration tests. It's surprising how valuable these techniques really are in pen testing, given how easy they are to do. In this article, Yori focuses on looking for communication streams. --Ed.]

By Yori Kvitchko

There is a mentality common to many software developers that we as penetration testers should always relish and keep in mind; if some feature I need to call or use is packaged inside of a self-contained individual executable file, it is safe, trustworthy, and out of sight. As a software developer myself, I often have to catch myself on this same line of thinking. Just because it's not easy to get to, doesn't mean that someone won't. Keeping this in mind, we as penetration testers shouldn't be afraid to break down the walls of an executable or app and get at its juicy innards.

Lets face it though, to some penetration testers, reverse engineering is a dark art that just feels out of reach, a practice only for the ninja masters. It's no simple task to move from web application analysis to following jump calls in IDA Pro. But, even before you've achieved ninja masterhood, there are a lot of practical things you can do to tease out important secrets and perhaps even vulnerabilities in stand-alone application binary files.

This set of blog posts is going to present a few reverse engineering techniques that I've found to be particularly useful over the last few years. Most importantly though, these are techniques that anyone can learn and use quickly and easily.

One of the juiciest types of applications you can encounter on a penetration test is internal tools. These tools, typically written with a "just get it working" mentality, often have shoddy security because no developer of internal tools thinks about an external threat getting access to them. We, of course, know that these tools are just a pivot away from being in our grasp. Here are some situations to be on the lookout for when you might have an opportunity to use these quick and useful binary analysis techniques:

- Any time you've compromised a machine with a standalone internal application installed, often an internal tool on a client machine. Be on the lookout for anything branded with the company's name.
- When target system personnel ask you to test the application as part of your penetration test.
- After getting access to an internal web server that has a Flash/Java/other client side application embedded into the hosted website.

- Any other time that you find a binary or application that can be considered in scope. Make sure to check shared drives and ftp servers for folders named "tools" or similar.

  Dealing with a binary, it is often the communication channel that the binary uses that is most vulnerable to attack. So how do we figure out who the binary, Adobe Flash file, or other standalone application is communicating with? Many developers, due to its ease of use, choose to use HTTP as a form of sending data from and to an application. Keeping this in mind, the first thing I do when I analyze an application is run strings on it looking for HTTP or HTTPS URLs.
  Linux:

  ```
  $ strings app.exe | grep -E -i 'https?://'
  ```

  Windows (requires the Strings utility from Microsoft Sysinternals):

  ```
  C:\> strings app.exe | findstr /i /r "htt[ps]*://"
  ```

  Nowadays, lots of standalone applications are nothing more than fancy frontends for a regular old web application. If you determine that there's a web server behind the application, your job just got that much easier. For some extra fanciness, take a memory dump of the running application (using a tool such as mdd or one of the other tools listed here) and run strings on that. You might find some crafted GET or POST requests with passwords or other interesting parameters.
  Even if the application doesn't use HTTP, there are plenty of easy-to-analyze-and-manipulate data formats that the application might be using to communicate over the network. Before we analyze any network protocols though, we need to capture the traffic being sent by/to the application. We'll be focusing on Windows here since most of the juicy applications we want to analyze will be Windows client applications.

  Fire up the application and find its PID. You can do this with Task Manager, Process Explorer, or the plain old command line as follows:

  ```
  C:\> tasklist /FI "IMAGENAME eq app.exe"
  ```

  Once we have the PID, lets say it's 6140, we can look at which IP addresses the application is communicating with. Open up your command prompt and run netstat, displaying PIDs and re-running every 1 second.

  ```
  C:\> netstat -na -o 1 | findstr "6140"
  ```

  If nothing immediately appears, try interacting with the application to get it to make an outgoing connection or open a listening port. Once you have an IP address, a simple Wireshark filter finishes up the job for us, isolating just the IP address we're interested in.

  ```
  ip.host == 10.10.10.1
  ```

You can also look for DNS requests that were used to get that IP address with the following filter.

```
dns.resp.addr == 10.10.10.1
```

DNS names can sometimes offer clues as to the nature of the back-end. Don't forget to do an Nmap version scan on the port as well.

Once you have an isolated communication channel, use Wireshark's "Follow TCP Stream" and look for easily readable and easy-to-manipulate data formats. Some things to keep an eye out include HTTP, Base64, XML, CSV and any other plain-text protocol. The rest depends on how well you can take advantage of this new communication channel. Remember that you take advantage of proxies (like the OWASP ZAP Proxy or Burp's proxy) for HTTP and also forge your own communication with plain old Netcat. The rest is up to your imagination and the lack of imagination on the part of the developer.

In the next episode of this series, I'll provide additional quick and dirty tips for penetration testers in analyzing stand-alone binary files to pry out useful secrets.

--Yori Kvitchko
Counter Hack

16 May **2013**

Part 2: Quick and Useful Tricks for Analyzing Binaries for Pen Testers

0 comments Posted by eskoudis
Filed under Uncategorized

[Editor's Note: In his previous blog post, Yori Kvitchko provided a bunch of tips penetration testers could use to analyze binary files, focusing on network communications. This time around, Yori looks at application data files, a hugely important source of information that could include passwords, hashes, or other sensitive stuff leaking out of an application. The techniques Yori describes here are some important building blocks for all pen testers to apply to the applications we analyze. --Ed.]

by Yori Kvitchko

This blog post is the second in a series of three blog posts dedicated to quick and useful techniques for analyzing binaries. In my first post, I talked about how penetration testers and other analysts can find and isolate network traffic generated by a binary. This time we'll look at pillaging the various data files that binaries and applications leave lying around. Our focus will again be on the Windows side of things, as that's where we often find the juiciest applications to analyze, including client-side software and internal tools.

In Windows, application data files are often placed in the AppData folder. This folder, specific to each user, can be a gold mine of information about installed applications. Located in each users' home directory, AppData (Application Data on Windows XP) can be accessed using the %AppData% environment variable, as in "dir /a %AppData%" (I put in the /a after the dir command to show folders with any combination of attributes; sometimes items in the AppData directory have a hidden attribute, so we want to see all them with dir /a).

Here are some useful files to keep an eye out for:

- History, Cache and Temporary Files — The prime example here is anything related to a web browser. Many applications take advantage of Internet Explorer for browser windows inside of the application, so IE cache and temporary files can prove insightful about the activities of the user and may hold sensitive, leaked information.
- Configuration Files — Anything from IP addresses to passwords might turn up here.
- Saved Data Files — If the application in question has a saved data file format, these saved data files can often contain useful information in the data as well as the meta-data.

To analyze these application files, as well as any files found in the application's main directory, we can employ a number of different techniques. First and foremost, our good friend, the strings command, proves to be just as useful in analyzing data files as it is on binary files. Be on the lookout for the same type of data we talked about in the first blog post, URLs and IP addresses especially. Strings is freely downloadable for Windows fromMicrosoft Sysinternals. It's also worthwhile noting that the Sysinternals strings looks for both ASCII and Unicode strings by default, returning sequences of three characters or more in length.

Once we have some files on our hands, and we've done some cursory analysis with strings, we've likely

either encountered a cleartext ASCII file (XML is especially common!) or we have a binary data file on our hands. If that's the case, we have two easy-to-use sources of information to help us identify the type of file. The file extension itself (such as .asc, .xml, or .docx) and the Linux "file" command. The file command can also be downloaded for Windows from GnuWin32. We can use it as follows:

```
$ file data.ext
data.ext; gzip compressed data, from Unix, last modified: Mon Mar 11
14:35:58 2013
```

Let's take a look at some of the more common data file formats we run into and how to extract data from them.

zip/tar/gzip — Decompress with the zip program of your choice. These files usually contain another data file inside. Many data formats are actually zip files in disguise. It's always worth trying to rename a file to ".zip" and try to unzip it just to check.

plist — Property list files used by Mac OS X and iOS. These files are especially useful when analyzing a mobile app. You can use a tool like plist Editor (http://www.icopybot.com/plist-editor.htm) to view the contents of the file.

sqlite — Sqlite databases are sometimes used for local data storage (as in Firefox and many mobile applications). Use SQLiteSpy (http://www.yunqa.de/delphi/doku.php/products/sqlitespy/index) or the Linux tool sqlite3 to view the contents of the database.

swf — Not exactly a data file format, but swftools (http://www.swftools.org/) does a great job of extracting images, sound, video, and even source code from SWF Flash files.

General Metadata — For general metadata, not specific to any file type, use ExifTool (http://www.sno.phy.queensu.ca/~phil/exiftool/) to see all metadata attached to a file. Images and documents are especially good sources for valuable metadata.

File System Images — Although not strictly a data file created by applications, we as security practitioners often have the need to work with file system images and some developers can occasionally get pretty crafty. Check out AccessData's FTK Imager (http://www.accessdata.com/support/product-downloads) for mounting all manner of file systems.

If the file format isn't in the above list, Google is a great resource for finding tools to analyze specific file types. If all else fails, see if a local installation of the application that created the file can be used to read it and extract its data. A great example of this would be copying a Firefox profile locally and importing into a local Firefox installation to view cookies and history.

In our next (and last) installment of this blog post series, we will look at some simple techniques for decompiling executables and DLLs to give us that last bit of insight we might need. After all, what could be more juicy when analyzing a binary than being able to peer right at the source? Until then... signing off --

--Yori Kvitchko
Counter Hack

07 Jun **2013**

Part 3: Quick and Useful Tricks for Analyzing Binaries for Pen Testers

0 comments Posted by eskoudis
Filed under Uncategorized

[Editor's Note: In part 3 of this series on techniques penetration testers can use to analyze executable files, Yori Kvitchko takes a look at reverse compiling code, with specific tips for Python and Java. They are often chock full of useful stuff in pen testing, and Yori provides a bunch of helpful tips in teasing out their secrets! --Ed.]

By Yori Kvitchko

In the first part of this series, I discussed analyzing binary files and looking for hints about their communications streams. In the second part of the series, I delved into the data files that binaries often create. For the third and final blog post in this series about analyzing binaries, I'll be discussing some quick and easy techniques for decompiling binaries that don't compile into raw assembly. I use the term "compiled" here rather loosely. The extent and difficulty of analyzing assembly is outside the scope of this blog post, so what I'll be talking about is analyzing executable files that are not pure assembly. These executable are either a "compiled" version of an interpreted language, such as Python, or are compiled into byte code and run in a virtual machine such as the Java Virtual Machine (JVM).

Seeing the source code of an application is big. It's the grand slam of reverse engineering because it gives you pretty much everything about what's going on inside the program itself. If you can see the entire source code of an application, assuming no trickery such as staged downloads, you know everything the application does. Just having the source code by itself isn't very helpful though, unless you know what to look for. Not only can source code often be difficult to search through to find what you're looking for, but it also depends on you knowing the language it's written in well enough to analyze it. So, what are some easy things to look for in source code that don't require poring over each line of code?

The usual suspects from my last two blog posts are much easier to look for in source code. Hardcoded passwords, URLs, and even XML or plain text configurations can often be found by searching for with relevant keywords that often appear in variable names such as "pass" or "xml". On top of that, looking at the source code can also inform the process of analyzing files created by the executable. Often times, encoding or encryption is done with libraries that can easily be seen in source, telling us exactly how we might decode a given file. Furthermore, looking at source code is probably the easiest method for decoding a network protocol. Searching for function names such as "getInt" or "getString" can often reveal the area of the source responsible for decoding a custom protocol.

Now that we know what to keep an eye out for, here are some languages and corresponding tools that will let us get at the juicy source code innards of these executables. Keep in mind though, that there are source code obfuscation tools for each of these languages that can make an executable much harder to analyze with these techniques.

C# .NET

C Sharp is everywhere; from standalone applications, to languages inside of other tools, to web applications. Microsoft's .NET platform and all related technologies are flexible, relatively easy to code, and therefore used all over the place. With such a large attack surface to work with, as well as a similar need coming from the developer side, it's no wonder that there are a number of tools that can decompile C# binaries and libraries.

Probably the best of these tools is redgate's Reflector. It works exceptionally well, rarely fails, and has a number of great features that help in analyzing the resulting source code. The only problem is that it isn't free. If you need it though, it might be well worth the asking price.

On the free side, we have ILSpyas one of the top contenders for decompiling and analyzing C#. In addition to having a relatively high success rate in decompiling code, it also has a number of very useful features. First, it allows you browse the code much like you would in a modern IDE. Going to the definition of a function or variable is easy, as is finding all references to them. On top of that, ILSpy allows you to save all of the relevant source code as a series of source files and corresponding project file. This gives you the ability to then browse the source in the editor or IDE of your choice and use tools such as grep to quickly search through it.

Both of these tools will automatically analyze any imported libraries making it easy to see what the analyzed source code is importing and browse through that code as well. I've personally used ILSpy to great effect, while trying to replicate a network protocol.

Java

Although losing its popularity a bit in the last few years, Java still has a number of developers dedicated to it and the advent of Android phone development has only strengthened Java's presence. Much like C#, Java is a byte code compiled language and can often be reverted back to near-original source code. Applications often come packaged in JAR packages, which are little more than zip files of byte code compiled Java code.

With most of the functionality provided by ILSpy for C#, the Java Decompileris freely available and does a great job with decompiling Java class files and JAR packages. It allows you to browse the source, find definitions, and save the source code for viewing in an IDE like Eclipse.

Probably the most common use for decompiling Java, other than internal tools, is the analysis of Android applications. For more on this topic, check out SANS Sec 575 which covers analyzing Android apps in detail.

Python

Ever the favorite for small scripts and the occasional Django web application, Python can still be found in the toolkit of many a developer. As an interpreted language, Python has an even looser definition of compiling, but still has a compiled layer in the form of Python byte code stored as ".pyc" files. These pyc files tend to be much more easily reversible than Java and C# so unless the source has been obfuscated you can almost always retrieve the exact source code made to create the final executable.

There are a number of tools that can perform the necessary decompile operation, but my personal favorite is depython.com. It's easy and it works, but it only supports up to Python version 2.6. For later versions, check out unpyc and unpyc3.

Although decompiling is great, there is actually an even cooler alternative for getting at the innards of a Python script. Because Python is an interpreted language, interacting with it can be done in a much more dynamic fashion. For example, the python interpreter can be executed with the "-i" option. What this option does is executes the given Python script then, without clearing any state, immediately gives you a Python prompt. This prompt then allows you to run any Python command inside of the environment left over by the Python script. What this means is that any variables stored by the script can be accessed with a simple print statement, and any functions can be called at will. Both are discoverable and listed by using the dir() function. That's quite a bit of power just from using the built in features of the Python interpreter.

```
$ python -i someprogram.py
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'secret_variable']
>>> print secret_variable
secret value
```

That's it for my series on analyzing binaries. Any other languages I didn't cover will typically fall into the byte code category such as C# or the interpreted category such as Python, so a bit of Googling will often reveal tools that fill a similar function for those languages.
I hope those of you who are new to the subject learned a little bit to get started with and bypass that mental barrier of thinking that reverse engineering requires you to know assembly. For any experts reading, I hope you picked up a trick or two as well. Thanks for reading and as always, happy hacking!

--Yori Kvitchko
Counter Hack

References:
http://pen-testing.sans.org/blog/pen-testing/2013/04/18/quick-and-useful-tricks-for-analyzing-binaries-for-pen-testers
http://pen-testing.sans.org/blog/2013/05/16/part-2-quick-and-useful-tricks-for-analyzing-binaries-for-pen-testers
http://pen-testing.sans.org/blog/pen-testing/2013/06/07/part-3-quick-and-useful-tricks-for-analyzing-binaries-for-pen-testers