JULY 2012

Fighting malware and spam

CONTENTS

2 **COMMENT**
Where should security reside?

3 **NEWS**
Largest international carding crimes operation: 26 arrests
Hotel group fined
VB welcomes

3 **MALWARE PREVALENCE TABLE**

MALWARE ANALYSES
4 Noteven close
6 Tiny modularity

9 **FEATURE**
Malicious PDFs served by exploit kits

TUTORIALS
11 Unpacking x64 PE+ binaries: introduction part 1
20 Quick reference for manual unpacking II

23 **END NOTES & NEWS**

IN THIS ISSUE

MOVE CLOSER

As computing devices become more or less disposable, or quicker to reset and recover than to repair, what is it that we actually need to secure? Greg Day believes that, in the future, security must move closer to the information.

page 2

BUGGY CODE

Code virtualization is a popular technique for making malware difficult to reverse engineer and analyse. W32/Noteven uses the technique, but has such a buggy interpreter that it's a wonder the code works at all. Peter Ferrie has the details.

page 4

SMALL BUT MIGHTY

Researchers have found a small piece of malware capable of doing just as much as its bigger brothers. Raul Alvarez looks at the structure of the malware, its code injections and modular execution, and describes how the tiny 'Tinba' is capable of doing so much.

page 6

ISSN 1749-7027

vb

'It seems logical that, in the future, security must move closer to the information.'

Greg Day, Symantec

WHERE SHOULD SECURITY RESIDE?

In 2011, we finally saw some of the long predicted growth in mobile threats. While the numbers are still infinitesimal – thousands as opposed to the hundreds of millions of threats discovered on PCs last year – mobile device security is now a topic of discussion in the boardroom.

With this in mind, there is a question that I would like to hear discussed more widely: what is the right way to manage mobile threats?

The cost of a laptop computer has dropped significantly over the last decade[1]. It is predicted that the cost of smartphones will also drop by a third over the next couple of years[2], with some substantial decreases having already been seen this year (take RIM dropping up to 26% off some of its devices[3] for example).

But what is the relevance of the cost of devices? I increasingly find myself contemplating whether we will reach a point where the value of the device means that it simply isn't worth protecting. I'm not in any way suggesting that we no longer need security. My question is: what are we actually trying to achieve?

[1] http://blogs.computerworld.com/18748/how_low_can_they_go_laptop_prices_continue_to_drop.

[2] http://articles.businessinsider.com/2012-02-29/tech/31109577_1_smartphones-pc-sales-internet.

[3] http://news.telecomseva.com/index.php/2012/03/research-in-motion-decreases-smartphone-prices-by-up-to-26-per-cent/.

Editor: Helen Martin
Technical Editor: Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Simon Bates
Sales Executive: Allison Sketchley
Web Developer: Paul Hettler
Consulting Editors:
Nick FitzGerald, *Independent consultant, NZ*
Ian Whalley, *Google, USA*
Richard Ford, *Florida Institute of Technology, USA*

When I started working at *Dr Solomon's* in the early 1990s, recovering from a virus on a PC was a significant undertaking. Imaging was not common, and back-ups were poor, so systems would need to be built again from the ground up, and data could be lost permanently. We sold anti-virus software to mitigate the cost and effort involved in recovering from infection.

With a modern smartphone, it is possible to reset both the device itself and, in most instances, the apps installed. Increasingly, it will also be possible to restore data through services such as *iCloud*[4]. Furthermore, smartphones typically have a lifespan of just six to nine months from a manufacturer's standpoint[5], and most providers' contracts generally last from a year to 18 months. Given these facts, is it possible that the device has become a disposable shell that can be reset or replaced more quickly and cheaply than actually removing the infection/attack?

I increasingly wonder whether the existing security model is the right approach going forward.

As the device is becoming more or less disposable, or quicker to reset and recover than to repair, what is it that we actually need to secure? In the world of Social Mobile Cloud and information explosion, it seems that the two most pivotal factors are the integrity and confidentiality of the information we hold and use.

The disposable nature of the smart device and the resilience of the cloud go a long way towards ensuring availability. As a result, the priority is less about keeping the device up and running and more about keeping the information available and secure.

Whilst we still look to innovate with concepts such as security in the hardware, it seems logical that, in the future, security must move closer to the information. This means better integration with the vast array of information structure types. Take a look at the likes of *Google*, which is perceived to be the leader of Internet information management; it has made a number of security acquisitions as it recognizes the significance of security at the information level.

The mobile threat landscape and the most effective way to secure against it is undoubtedly a discussion that is here to stay. In my opinion, security needs to be as innovative as the devices, and needs to put information, rather than just the device, at the forefront.

[4] http://www.apple.com/icloud/.

[5] http://www.theusdvista.com/business/android-os-changes-smartphone-life-cycle-1.2000033.

NEWS

LARGEST INTERNATIONAL CARDING CRIMES OPERATION: 26 ARRESTS

The results of what federal officials are calling the largest ever coordinated international law enforcement action directed at carding crimes were revealed last month when the US Justice Department released documents detailing a two-year operation in which FBI officials set up an undercover carding forum (carderprofit.cc) to catch users buying and selling stolen credit card details. During the operation the Justice Department said it passed information on to financial institutions regarding more than 411,000 compromised credit and debit cards, and notified 47 companies, government entities, and educational institutions about breaches of their networks. A total of 26 people were arrested under suspicion of trading in stolen credit card accounts.

HOTEL GROUP FINED

Large hotel group *Wyndham* has been fined by the FTC for data breaches that resulted in the loss of hundreds of thousands of customers' confidential data.

The FTC claims that *Wyndham* failed to maintain 'reasonable security' on its networks, thus allowing a series of data breaches to occur.

According to the FTC's complaint, the hotel group failed to adequately protect a property management system used to manage 7,000+ hotels under the *Wyndham Hotels and Resorts* umbrella. Among other things, it is believed that default administrative usernames and passwords were used on servers that connected to the network.

In addition, *Wyndham Worldwide* – the hotel group's parent company – stored customer credit card data in plain text, and did not adequately segregate the property management system from the company's intranet and the public Internet. As a result, a string of security breaches occurred between April 2008 and January 2010, and customer data was stolen.

The company says it has improved its information security practices and that it plans to challenge the suit.

VB WELCOMES

Virus Bulletin welcomes a new member of staff this month. Tom Gracey has joined the *VB* team as Perl Developer. His work will focus mainly on updating and maintaining the *VB* website, but he will also assist with some aspects of the VBSpam and VB100 tests – which we hope will allow us to introduce a new set of tests for URL filters later in the year. Tom and the rest of the team will be at VB2012 in Dallas in September: http://www.virusbtn.com/conference/vb2012.

Prevalence Table – May 2012[1]

Malware	Type	%
Autorun	Worm	11.71%
Conficker/Downadup	Worm	6.60%
Downloader-misc	Trojan	6.45%
Iframe-Exploit	Exploit	6.42%
Heuristic/generic	Virus/worm	5.37%
Injector	Trojan	3.74%
Crypt/Kryptik	Trojan	3.58%
Java-Exploit	Exploit	3.48%
Heuristic/generic	Trojan	3.30%
Adware-misc	Adware	2.92%
Blacole	Exploit	2.87%
Sality	Virus	2.82%
Agent	Trojan	2.60%
Sirefef	Trojan	2.58%
FakeAV-Misc	Rogue	2.28%
LNK-Exploit	Exploit	1.77%
Dorkbot	Worm	1.68%
Banload	Trojan	1.56%
Jeefo	Worm	1.47%
Virut	Virus	1.37%
AutoIt	Trojan	1.27%
Exploit-misc	Exploit	1.19%
Crack/Keygen	PU	1.07%
Ramnit	Trojan	1.03%
Dropper-misc	Trojan	1.01%
Encrypted/Obfuscated	Misc	0.97%
PDF-Exploit	Exploit	0.96%
Wimad	Trojan	0.84%
Delf	Trojan	0.74%
Redirector	PU	0.70%
Brontok/Rontokbro	Worm	0.68%
Phishing-misc	Phish	0.66%
Others[2]		14.32%
Total		**100.00%**

[1]Figures compiled from desktop-level detections.

[2]Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

MALWARE ANALYSIS 1

NOTEVEN CLOSE

Peter Ferrie
Microsoft, USA

Code virtualization is a popular technique for making programs difficult to reverse engineer and analyse. While its use is seen mainly in commercial products such as *VMProtect*, some viruses use the technique for the same purpose. The interpreter is the weak point in any virtualization implementation, because if the interpreter can be understood, then the virtualization can be reversed to some degree. In some cases, the interpreter is intentionally made difficult to read. In the case of the W32/Noteven virus, the interpreter is difficult to read due to sloppy coding and numerous bugs. We can safely assume that this is unintentional.

UNSTRUCTURED EXCEPTIONS

The interpreter begins by installing a Structured Exception Handler to point to a location within the interpreter. There is a bug in this code, which is that the virus breaks the SEH chain, so any code that needs to walk the chain will fail and possibly cause an exception that will result in the program being terminated. The interpreter retrieves the address of a system DLL by walking the InInitializationOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. This results in the interpreter retrieving the address of kernelbase.dll on *Windows 7* and later versions, but kernel32.dll on earlier versions of *Windows*. The interpreter resolves the addresses of the functions that it needs in order to infect files (FindFirstFile, FindNextFile, CreateFile, GetFileSize, ReadFile, WriteFile, CloseHandle, VirtualAlloc, VirtualProtect and GetCurrentDirectory [which is unused]). Fortunately for the virus author, all of these functions are present in kernelbase.dll.

The interpreter allocates space on the host stack for the encoded instructions. There is a minor bug in this code, which is that depending on the size of the instructions, the resulting stack pointer value might be misaligned. Fortunately for the virus author, the interpreter uses no APIs that require the stack to be aligned (such as FindFirstFile – which is called by the virtualized code, and the interpreter merely resolves it on behalf of the virtualized code). The interpreter copies the array of instruction lengths to the host stack for processing later, then allocates two blocks of memory. The first block is the stack for the virtualized code, and the second block is the virtual machine buffer that will hold the virtualized code. The interpreter then copies the virtualized code to the virtual machine buffer, and unmaps the virtual machine buffer in order to 'protect' it from being read externally (though nothing stops the original copy from being read instead).

The interpreter swaps to the virtual machine stack, saves the CPU flags there, allocates space for the virtual machine registers, and then swaps back to the host stack. The interpreter copies the current register values to the virtual machine stack, and then begins parsing the virtualized instructions using the instruction lengths that were copied to the host stack earlier. The parsing is performed in the reverse direction, for no obvious reason. The interpreter maps the virtual machine buffer, copies the virtualized instruction from the virtual machine buffer to the virtual machine stack, according to instruction length, decodes the instruction, and then unmaps the virtual machine buffer again.

WHAT'S IN A NAME?

The decoded instruction is the original native CPU instruction. The virus makes no attempt to transform the opcodes in any way. Each instruction is examined before execution, because the interpreter will perform a controlled execution of the 'safe' instructions. Certain instructions cannot be executed directly because they will cause a loss of control and possibly a crash. This means that the environment is not code virtualization in the typical sense, but rather buffered code execution. The difference is not important for the purpose of this article.

The instructions that are considered to be special by the interpreter are: E8 (call rel32), C3 (ret), FF (various), EB (jmp rel8), E9 (jmp rel32), 0F 80-8F (jcc rel32), 70-7F (jcc rel8) and E0-EF. There is a bug in the last set, which is that it should be restricted to E0-E3, but the way in which the comparison is made throws away the entire low nibble instead of just the low two bits. This prevents the interpreter from supporting the missing instructions and can result in unexpected behaviour.

If the instruction is not considered to be special, then the interpreter appends a call/fault/jmp sequence before allowing the instruction to run. The call is used to save the instruction pointer onto the stack. The faulting instruction is used to transfer control back to the interpreter. The jump is supposed to transfer control back to the interpreter in the event that somehow the faulting instruction did not do so, but the jump offset is completely wrong, so if the jump were ever hit, then the virus would crash. The interpreter intercepts the exception and ensures that it is the expected kind. The faulting instruction is an interrupt 3 instruction, which can interfere with a debugger and make the code difficult to trace. The context that is saved when an exception occurs will be used to update the registers in the virtual machine.

In order to execute an instruction, the interpreter saves the host registers on the host stack, swaps to the virtual machine

stack, restores the virtual machine registers from the virtual machine stack, and then runs the instruction. When the interpreter intercepts the exception, it reinstalls the broken Structured Exception Handler vector and installs another Structured Exception Handler to point to a location within the interpreter. It also saves the virtual machine registers on the virtual machine stack, swaps to the host stack, and then restores the host registers from the host stack.

CALL ME CRAZY

If the instruction opcode is 'E8', then the interpreter checks if the relative offset is within the limit of the virtual machine buffer. There are two bugs in this routine. The first is an off-by-one boundary condition, which allows the call to land on the first byte beyond the end of the buffer. The second bug is an off-by-n boundary condition, which does not require that the next instruction to execute fits in the space remaining in the buffer. The interpreter also implicitly limits the size of the virtualized code, so anything outside of the buffer is treated as though the relative offset were zero for the call. If the instruction is accepted, then the interpreter inserts space for a dword on the virtual machine stack. There is a bug in that routine, too, which is that there is no check against the stack limits. As a result, a stack fault will occur when the stack becomes full. Otherwise, the interpreter places the return address in the newly created space, updates the stack pointer, and determines the new instruction pointer location. There is yet another bug in this code, which is that a missing instruction prevents the interpreter from supporting calling backwards in the code. Not only is it not supported, it is also misinterpreted – any attempt to call backwards will be treated as a call forwards by the absolute relative value (that is, a call backwards by six bytes will become a call forwards by six bytes).

POINT OF NO RETURN

If the instruction opcode is 'C3' then the interpreter fetches the return address from the virtual machine stack, deletes the element from the virtual machine stack, and adjusts its position in the length array corresponding to the return address. There is a bug in this code, which is that the interpreter assumes that it will be returning to an earlier position in the array. Any attempt to return to a later position will cause the virus to crash.

If the instruction opcode is 'FF', then the interpreter checks if the eax register contains certain values. One value retrieves the in-memory address of the interpreter. Another causes a return of control to the host entry point, however there is a bug in the state that is passed to the host. Another value retrieves a pointer to the list of resolved API addresses. If the value is none of these, then the interpreter attempts to run the instruction as described above. There is a bug in this behaviour, which can result in escape from the virtual environment, because the interpreter does not prevent the interpreted code from jumping to an arbitrary address.

If the instruction opcode is 'EB', then there is a bug: the interpreter forces an exception to occur, perhaps for debugging purposes, but since the wrong stack is in use at that moment, the result is that another exception occurs. The second exception is intercepted by the interpreter, but during the handling, another exception occurs. This cycle repeats until a stack fault causes *Windows* to terminate the program.

If the instruction opcode is 'E9', the interpreter calculates the new instruction pointer and continues execution.

If the instruction opcode is '0F 80-8F' or '70-7F', then the interpreter attempts to simulate the branch. The way in which this is done is about as non-optimized as it is possible to be. Instead of simply loading the flags and then replicating the branch instruction locally, the virus examines the flags according to the conditions that they are supposed to represent, and then branches to a unique label for each of the true and false conditions. Of course, there is also a bug in this code. The 'jbe' simulation has its conditions reversed, so a branch is taken when it shouldn't be.

If the instruction opcode is 'E0-EF' then we encounter more bugs. The first is that the handling for the 'E0-E2' set (LOOPNE, LOOPE and LOOP) skips the instruction if the ecx register is zero. A real CPU will perform the loop as though it had an initial iteration count of 4GB. That is, the value of zero in the ecx register is not special. Only a value of one can cause a loop to exit immediately, and the value in the ecx register is altered in all cases. The routine is broken anyway, because it forces an exception to occur, and demonstrates the recursive exception problem described above.

The interpreter also attempts to intercept faults in the virtual machine buffer, but there is a bug in this code: the interpreter uses the wrong offset for fetching the faulting address from the Exception Record. This results in a crash in the interpreter while it is searching the length buffer for the exception address. This bug also demonstrates the recursive exception problem described above.

VIRTUAL MALWARE

The virus that the interpreter runs begins by requesting the API addresses from the interpreter. The virus copies them to a local buffer, and caches the FindFirstFile, FindNextFile, and CreateFile addresses in registers, but the values are lost during a memory allocation that follows immediately. The caching might have been for debugging purposes to

see the values, because the addresses are loaded again later, as needed. The virus enumerates the objects in the current directory, but discards the first two results, and begins examining the objects from the third one that is found. There is a minor bug in this idea, which is that the virus might miss some files whose names cause the directory sorting to place them before the '.' and '..' directories. The virus attempts to check that the extension is '.exe', but forgets to check for the '.', so a file named 'exe' is also accepted.

The virus attempts to open the file, allocate memory to hold the entire file plus another 4KB, read the whole file, and then close it. The only operation whose result is checked is the file open. The virus assumes that the other operations will succeed, or that the exception handler in the interpreter will intercept any problem (but, as we have seen above, this is not the case). There is another bug in this code, which is that neither the interpreter nor the virus frees any of the memory that they allocate. If enough large files exist in the directory, then eventually the process will be terminated by *Windows* due to memory exhaustion. The virus skips the file if it is infected already. The infection marker is the value 'EEEE' stored in the OEM ID space in the MZ header.

The infection routine contains numerous bugs, the most obvious of which is that the virus does not check the Machine field when infecting files, so 64-bit files will be corrupted. The virus does not check other important fields either, so DLLs and drivers will be infected, too. The virus marks the last section as writable but not executable, so it will fail to run on DEP systems if the last section was not already marked as executable. The virus saves the absolute address of the host entry point in the virus body, so the virus will fail to run the host if the host supports ASLR.

The virus fetches the Load Configuration Table data directory information, and assumes that if it is present, then it is located in the first section. This can result in a pointer to an unexpected location in the file. The virus attempts to zero out the actual table, instead of the data directory entry. The virus appends itself to the last section and changes the host entry point to point to the interpreter code. It then attempts to open the file, write itself, and close the file again. The virus does not check that the file attributes allow the file to be written to, and it does not do anything with the file's date and time stamps or the checksum. Files with appended data will have their appended data overwritten by the virus code. Once the virus has finished enumerating the files, it returns control to the host.

CONCLUSION

Analysing virtualized code can leave us wondering how it works. In this case, we're left wondering how it works, given how buggy it is.

MALWARE ANALYSIS 2

TINY MODULARITY

Raul Alvarez
Fortinet, Canada

Researchers have found a small piece of malware capable of doing just as much as its bigger brothers. Dubbed 'Tinba' (Tiny Banker) [1], the malware is approximately 20KB in size. The size of the malware itself is nothing unusual – we have already seen malware of around the same size and even smaller. But, generally, smaller-sized malware tends only to perform very specific tasks, such as downloading components, creating backdoors, and other trivial things. What sets this one apart from the pack is its ability to do much more.

Using behavioural analysis, we can describe Tinba's main functionality. Using static analysis, we can predict and confirm what the code is doing. But it is only by following its footsteps that we can observe the modularity of Tinba's execution.

This article will look at the internal structure of the malware, its code injections, and its modular execution. We will describe how such a small piece of malware is capable of doing so much.

DECRYPTION ALGORITHM

The particular sample that we will look at is 19,968 bytes in size. It is small, but it has a simple encryption/decryption routine.

When the sample is executed, its initial routine starts with a simple decryption algorithm. After setting up the destination memory and the starting location of the encrypted bytes, it XORs each byte with 0xBF. A total of 13,312 bytes will be decrypted to the memory.

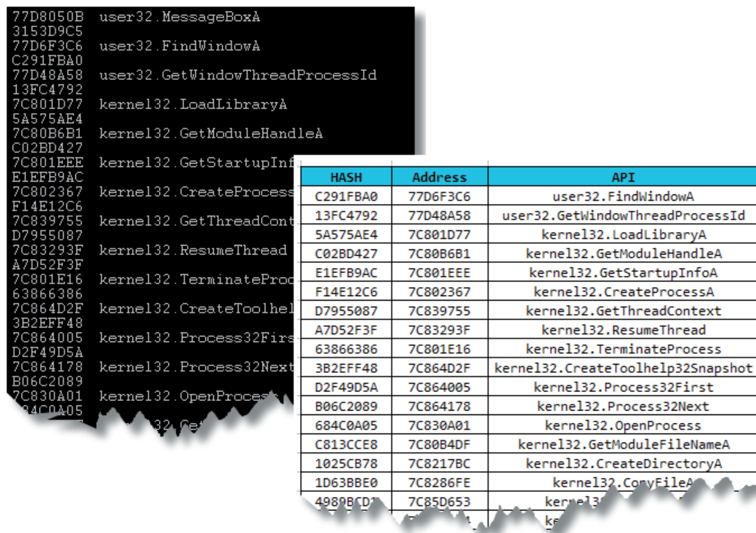
After decryption, the strings, texts and code that Tinba uses are visible, including the domain names it tries to connect to. After the decryption routine, the malware resolves the APIs that it needs to execute its other tasks.

THE HUNT FOR APIs

The APIs used by the malware are taken from five main DLLs: kernel32, ntdll, advapi, ws2_32 and user32. With the exception of kernel32, all of these names can be seen in the decrypted strings.

The image bases of the DLL names are used to resolve the APIs that Tinba needs for its malicious actions. These are acquired as follows:

First, the image base of kernel32.dll is acquired by parsing the Process Environment Block (PEB). The malware simply



HASH	Address	API
C291FBA0	77D6F3C6	user32.FindWindowA
13FC4792	77D48A58	user32.GetWindowThreadProcessId
5A575AE4	7C801D77	kernel32.LoadLibraryA
C02BD427	7C80B6B1	kernel32.GetModuleHandleA
E1EFB9AC	7C801EEE	kernel32.GetStartupInfoA
F14E12C6	7C802367	kernel32.CreateProcessA
D7955087	7C839755	kernel32.GetThreadContext
A7D52F3F	7C83293F	kernel32.ResumeThread
63866386	7C801E16	kernel32.TerminateProcess
3B2EFF48	7C864D2F	kernel32.CreateToolhelp32Snapshot
D2F49D5A	7C864005	kernel32.Process32First
B06C2089	7C864178	kernel32.Process32Next
684C0A05	7C830A01	kernel32.OpenProcess
C813CCE8	7C80B4DF	kernel32.GetModuleFileNameA
1025CB78	7C8217BC	kernel32.CreateDirectoryA
1D63BBE0	7C8286FE	kernel32.CopyFileA

Figure 1: A partial list of resolved APIs and their hash values.

looks for a match for '32', which is part of the name of kernel32.dll. No other checking is done to make sure the full name is 'kernel32'. Fortunately for Tinba, the first DLL with '32' in its name is kernel32.dll. Once it obtains the image base of kernel32.dll, it parses its header to look for the export table. Once the export table is located, the malware will get the hash value of each API in kernel32.dll and compare it with its own list of hash values (see Figure 1). This is the malware's method of resolving the API addresses without a regular call to the GetProcAddress API. After resolving the APIs that it needs, it moves on to get the image base of the next DLL.

The image base for ntdll.dll is acquired next, using a call to the GetModuleHandleA API using the string 'ntdll' as a parameter.

The image bases of advapi32.dll, ws2_32.dll and user32.dll are then acquired using the LoadLibraryA API with the string names of the DLLs as parameters.

The APIs the malware needs from these DLLs are resolved using the same method as for kernel32.dll.

FIRST CODE INJECTION

After the exhaustive resolution of APIs, Tinba creates a new process named 'winver' in suspended mode. It copies its 12,744 bytes of decrypted code to the winver process, and executes it remotely by calling the ResumeThread API. Afterwards, the original Tinba process is terminated. It is interesting to note that the malware hasn't done anything beyond decrypting its code and injecting it into the winver process.

Once Tinba has transferred execution to the winver.exe process, it starts the API resolution again in the same manner as it did during the execution of the original malware. This is to make sure that it gets the right DLL and the right addresses for the APIs. The code for API resolution is the first set of routines from the decrypted code.

Once the malware has determined that it is running in the context of the winver process, it looks for explorer.exe from the list of processes and injects its code in the same way as it did with the winver.exe process. This time, however, the CreateRemoteThread API is called to trigger the code in the explorer.exe process. This is the second code injection performed by Tinba.

This time the malware doesn't terminate the execution of the winver.exe process after the code injection into explorer.exe.

Tinba proceeds by initiating Internet connectivity for its communication with the C&C server. The routine to connect to the C&C servers is executed within the winver.exe process. The domain names are hard-coded and can be seen in the decrypted code of the malware (see Figure 2).

SECOND CODE INJECTION

Once the code injected into explorer.exe executes, Tinba performs the API resolution again, and checks to see which process it is running in. Once the malware has determined that it is running inside the explore.exe process, it performs the following familiar malware routines:

1. It checks whether the original sample is running as %APPDATA%\default\bin.exe. If it isn't, it creates the %APPDATA%\default directory, then moves the

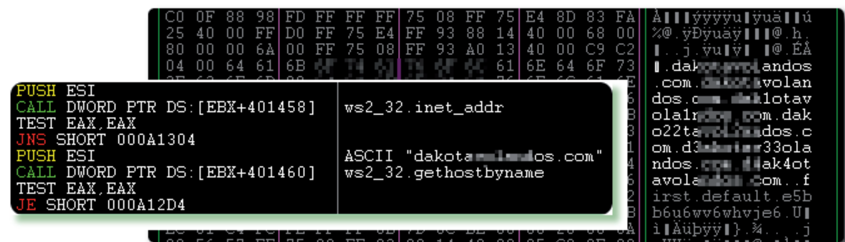


Figure 2: C&C domain names and the code that tries to access them.

original executable to %APPDATA%\default as 'bin.exe'. This, eventually, will look like it has deleted the original file and dropped a copy of itself.

2. It makes start-up registry keys with %APPDATA%\default\bin.exe, to make sure that Tinba will survive after a system reboot. Details of the registry keys are as follows:

```
Key: HKEY_CURRENT_USER\Software\Microsoft\
Windows\CurrentVersion\Run
Value: default
Data: "%APPDATA%\default\bin.exe"

Key: HKEY_USERS\[SID]\Software\Microsoft\Windows\
CurrentVersion\Run
Value: default
Data: "%APPDATA%\default\bin.exe"
```

3. The malware also modifies HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3 with a value of 1609, also known as DisplayMixedContentInternet.

4. It reads %APPDATA%\Mozilla\Firefox\profiles.ini, to get the default profile location. This is also one way to check whether *Firefox* is installed in the system. It looks for the 'Path=' from within the file, takes the '[profile code].default' value, and creates a folder path: %APPDATA%\Mozilla\Firefox\Profiles\[profile code].default.

   Afterwards, Tinba creates the file %APPDATA%\Mozilla\Firefox\Profiles\fzdq808c.default\user.js, which contains:

```
user_pref("security.warn_submit_insecure",false);
user_pref("security.warn_viewing_mixed",false);
```

   Effectively, this disables the security warning that appears when the user tries to send data through an insecure site, and disables the warning that appears when the user opens a page containing both encrypted and unencrypted data.

5. Then, it creates and runs a thread that does the following:

   It checks the list of processes for iexplore.exe, firefox.exe and chrome.exe. Once any of the three processes is found, it injects the 12,744 decrypted bytes again (giving us the third code injection), and executes the code by calling the CreateRemoteThread API. The code will only be injected into the browser process if the browser is not yet infected.

   After injecting its code into the available browsers, it goes back and lists the processes again, and checks whether any new ones have been added since the last time it checked the list. The thread keeps running to check for new browser processes to inject its code into.

THIRD CODE INJECTION

The code injected into the running browser is the same as that injected into the explorer.exe and winver.exe processes. This is the third and final code injection performed by Tinba. If any of the iexplore.exe, firefox.exe and chrome.exe processes are found on the list of running processes, Tinba will inject its code to any or all of them. It has a thread running within the explorer.exe process that monitors for the execution of the browsers. Even if the user terminates the browser, Tinba will be able to inject its code again once the browser application is executed, thanks to the persistent thread running inside the explorer.exe process.

Once Tinba determines that it is running within the browser, it executes the code used to intercept the user's online activities. Tinba now acts as a man-in-the-browser.

WRAP UP

On initial execution, Tinba's only goal is to decrypt the malware code and inject the decrypted code into the newly created winver.exe process. It transfers the malware execution to the winver.exe process and terminates itself. There are two main tasks that Tinba performs while in the context of the winver.exe process: it injects its code into explorer.exe, and connects to the C&C servers. Once execution has been transferred to the explorer.exe process, the familiar malware routines are performed, including dropping and creating files, adding registry keys, and injecting code into browsers. Tinba injects its code into running browser processes, but it leaves a thread running inside explorer.exe to monitor and make sure that it infects new browser processes. Finally, modular execution inside the browser monitors the user's Internet activities.

One trivial feature of Tinba is its ability to know which running process it has been injected into and perform the relevant necessary tasks. Tinba may be small, but its code has been optimized ingeniously. If a larger sized piece of malware had this kind of coding structure, we would expect it to be a considerable challenge. But, ingenious or not, security experts will always find a way to catch up with the malware.

REFERENCE

[1] Tinba. http://www.csis.dk/en/csis/news/3566.

FEATURE

MALICIOUS PDFS SERVED BY EXPLOIT KITS

Didier Stevens
Contraste Europe, Belgium

The Portable Document Format (PDF) is still a very popular vector with cybercriminals for infecting as many *Windows* machines as they can. Although the PDF language was not designed to allow arbitrary code execution, implementation and design flaws in popular reader applications make it possible for criminals to infect machines via PDF documents. Let us explore how this is possible.

FILE FORMAT

The PDF file format is composed of objects that define how pages should be rendered by reader applications such as the ubiquitous *Adobe Reader*. These objects are logically organized in a hierarchical tree structure. We have a catalog object at the root, and find page objects lower in the tree structure. These page objects refer to other objects to define text and images to be drawn upon an empty page.

Here, we come across the first example of how malware authors can tailor PDF documents to attack PCs. PDF readers like *Adobe Reader* need to support a large number of image formats that can be included in PDF documents. This support requires a huge code base that inevitably contains programming errors. In 2009, *Adobe* had to release new versions of *Reader* to fix bugs in the JBIG2 rendering algorithms. JBIG2 is an image compression standard supported by *Adobe Reader* – but *Adobe*'s JBIG2 decompression algorithms were found to contain buffer overflows. Malware authors discovered how to craft a specially designed JBIG2 image that would cause a buffer overflow in the decompression algorithm.

Exploit developers love to discover buffer overflows because they can often lead to arbitrary code execution. The type of buffer overflows that exploit developers search for are the ones that eventually lead to EIP (Extended Instruction Pointer) control. The EIP is a crucial register in *Intel* x86 microprocessors, because it points to the next instruction to be executed. When exploit developers can control the value of the EIP register via a buffer overflow, they can control which instructions will be executed, and thus achieve arbitrary code execution. But controlling the address to which the EIP points is only one element of an exploit. Another important element is being able to include instructions that the malware author wants to execute. In most malicious exploits, these instructions are shellcode that will ultimately download and execute malware. Including shellcode in the exploit is often tricky, but malware authors have found a quick and dirty solution: the JavaScript heap spray. When a malware author develops an exploit that achieves EIP control, he still needs to be able to plant shellcode in memory at the address pointed to by the EIP. Including this shellcode in an exploit that triggers the vulnerability can often be very difficult or impossible to achieve, because of the specifics of the vulnerability.

JAVASCRIPT

The PDF language supports a couple of programming languages, one of which is JavaScript. PDF readers like *Adobe Reader* include a JavaScript interpreter. When JavaScript code is embedded inside a PDF document, it will be executed depending on the type of action that is defined. One such action is the opening of the PDF document – meaning that the PDF reader will execute the embedded JavaScript code when the PDF document is opened. This in itself is not a security issue, as the JavaScript implementation in PDF readers like *Adobe Reader* is sandboxed. Programs written in this JavaScript version cannot access or modify resources of the underlying operating system such as files and registry entries. JavaScript support in PDF documents is designed to augment the rendering of those documents – for example by calculating totals in order forms – and is designed to prevent alteration of system resources. This means, for example, that malware authors cannot write a JavaScript program to drop a trojan.

But malware authors can use JavaScript to plant the necessary shellcode for their exploit. They achieve this with heap spraying: the script creates a string that contains the shellcode preceded by a NOP sled – a long sequence of NOP instructions. Then it creates a large number of copies of this string. Since JavaScript is an interpreted language, it uses a memory management structure (heap) to store its variables. Thus, creating a large number of copies of a string that contains shellcode effectively fills the heap with shellcode. (This is likened to spraying shellcode into the heap, hence the term ‘heap spray’.)

Finding a vulnerability (like the JBIG2 vulnerability) in the PDF language parser is an important step towards achieving arbitrary code execution, but there is another popular tactic: finding a vulnerability in the JavaScript parser. A well-known example is the util.printf vulnerability. Exploit developers discovered that they can take control of the EIP register by calling util.printf with a very long numerical argument (*Adobe* released a new version of *Reader* to address this vulnerability in 2008). An exploit for util.printf first uses JavaScript code to perform a

heap spray, then uses JavaScript to trigger the vulnerability in util.printf.

The two major exploit avenues present in malicious PDF documents found in the wild are: a JavaScript heap spray followed by the triggering of a vulnerability in the PDF language implementation, or the triggering of a vulnerability in the JavaScript language implementation.

As JavaScript heap sprays are so often found in malicious PDF documents, disabling JavaScript support in your PDF reader is often recommended as a mitigating action. Disabling JavaScript support in *Adobe Reader* means that JavaScript code embedded in PDF files is not executed. Remember that this course of action does not prevent PDF language exploits, but since they often rely on JavaScript heap sprays to plant shellcode, they ultimately fail when JavaScript support is disabled.

EXPLOIT KITS

JavaScript is not only an essential tool for malware authors developing PDF exploits, but it is also crucial for the operation of exploit kits. Exploit kits are sets of programs running on a web server that are designed to automatically infect clients. When a user is directed to a web server hosting an exploit kit, the exploit kit will serve the client with malicious PDF files, Flash files, Java files etc., all containing exploits specifically tailored to infect the machine of the unsuspecting user. The exploit kit serves many exploits to the client in the hope that at least one will be successful and take control of the targeted machine. PDF documents with embedded JavaScript code are particularly well suited for use in exploit kits, because they offer two important advantages: versatility and stealthiness.

A PDF document with embedded JavaScript code is a versatile tool for an exploit kit because it can serve many exploits inside the same PDF document and activate the one that is most likely to be successful. *Adobe*'s JavaScript implementation comes with a function to check the version of *Adobe Reader*: app.viewerVersion. This function returns the version number of the reader that has opened the PDF document and is executing the embedded JavaScript code. By using the result of this function, authors of malicious PDFs can design their JavaScript code to include several exploits and select the best one with a JavaScript ‘if’ statement. For example, if the version of *Adobe Reader* is 8.1.2, the JavaScript code for the util.printf exploit will be launched, but if the version of *Adobe Reader* is 8.1.3, then the JavaScript code for the Collab.getIcon exploit will be launched. Launching the JavaScript code for the util.printf exploit with version 8.1.3 or later is pointless, because the util.printf vulnerability was patched with the release of version 8.1.3.

Malicious PDFs produced by exploit kits not only use app.viewerVersion to determine which exploit to launch. Many features in *Adobe Reader* are implemented via plug-ins. These plug-ins are actually DLLs that are loaded into the *Adobe Reader* process whenever the functionality they implement is required. JavaScript in *Adobe Reader* is implemented with the ECMA Script plug-in (file Escript.api). Malicious PDFs can retrieve the version number of the loaded ECMA Script plug-in by enumerating plug-in array app.plugIns and reading the property version for the plug-in with property name ‘EScript’.

This versatility not only allows authors of malicious PDFs to tailor their JavaScript code to launch the most appropriate exploit for the version of *Adobe Reader* their file is running in, but it even allows them to target different readers with the same PDF document, provided the targeted readers support embedded JavaScript. For example, assume a malware author wants to target both *Adobe Reader* and *Foxit Reader* with the same malicious PDF. Both readers had a vulnerability in the util.printf method, but the details of the exploit for each are quite different. An exploit for *Adobe*'s util.printf implementation does not work for *Foxit*'s util.printf implementation, and vice versa. Hence the malware author needs to write JavaScript code to determine which reader opened his malicious PDF document and to launch the appropriate exploit (provided the version is vulnerable).

One method to determine which reader the JavaScript code is running in is to use a property or method that is only declared in one reader, and not in the other. For example, the Net.SOAP.wireDump property is declared in *Adobe Reader*, but not in *Foxit Reader*. When this property is accessed from JavaScript code running in *Foxit Reader*, an exception will be thrown, while with *Adobe Reader*, a boolean value will be returned. When an exception is thrown, it interrupts the running JavaScript code, but this can be prevented by catching the exception with a JavaScript try-catch statement. So, by inserting the Net.SOAP.wireDump expression inside a JavaScript try-catch statement and catching the exception, it is possible to determine which reader the JavaScript code is running in, and launch the appropriate exploit.

Exploit kit developers want to prevent anti-virus programs from detecting their exploits, so they develop kits that serve ever-changing exploits. Malicious PDF documents with embedded JavaScript code are particularly suited for this, as JavaScript can be used to obfuscate the code in an infinite number of ways. This is especially the case if exploit developers limit their malicious PDF documents to JavaScript exploits, because then all malicious code can be obfuscated.

OBFUSCATION

JavaScript obfuscation is a vast subject. New techniques appear all the time, making the task of anti-virus engine developers difficult. And with JavaScript code embedded in PDF documents, there are even more obfuscation possibilities. One popular way to obfuscate JavaScript code is to split it up into different parts. Inside a PDF document there are several ways to split up JavaScript code and store the different parts. PDF document annotations are often used to split up embedded JavaScript code. Annotations allow a user of a PDF reader to annotate the document he is reading. Annotations can be text, but also text highlights and other symbols. Annotations can be made invisible, so that a user can view the original document without annotations.

Invisible annotations are used by authors of malicious PDFs to store partial JavaScript code. These snippets of code are accessed from JavaScript code with the getAnnotations method, recombined with string concatenation and then executed via the eval function. The string concatenation code is often convoluted to add to the overall obfuscation of the JavaScript code.

One last obfuscation technique that deserves a mention is encryption. PDF documents can be encrypted for two reasons: for digital rights management and for confidentiality. When a PDF document is encrypted, the structure of the document remains unchanged – the structure is not encrypted, but the content is. This means that objects and their properties remain unencrypted, while the strings and streams stored inside objects (the actual content) are encrypted. PDF documents are encrypted with a key derived (amongst other elements) from a user password and a hashed owner password. The hashed user and owner password are stored inside the PDF document. If the user password is empty, the key can be completely derived from elements stored inside the PDF document, and thus the user does not need to provide a password to view the PDF document. In other words, a PDF document that is encrypted with a key derived from the hashed owner password (for DRM reasons, like disabling printing) is 'obfuscated' because of the encryption, but can be decrypted (hence viewed) without requiring a password. Anti-virus products that need to 'deobfuscate' such PDF documents need to be able to decrypt PDF documents.

CONCLUSION

Malicious PDF documents are used on a large scale to infect *Windows* PCs. This trend started several years ago, with mass mailings of malicious PDF documents, and is likely to remain popular for several years to come because of the versatility and stealthiness it offers to exploit kit developers.

TUTORIAL 1

UNPACKING X64 PE+ BINARIES: INTRODUCTION PART 1

Aleksander P. Czarnowski
AVET INS, Poland

The x86-64 architecture is taking over from IA32 CPUs – but this should not come as a surprise, especially since major operating system players have been supporting it for years already. Of course, malware authors are aware of this revolution and thus they target executable files running natively on AMD64-compatible architectures and operating platforms. One of the most complex (and flexible) executable formats in the 64-bit world is *Microsoft Windows* PE32+ (since the name is a bit misleading, we will refer to it as 'PE+' in the rest of this article). Due to the closed-source nature of *Windows*, the best and most advanced debuggers and anti-debugging techniques have been developed for the Win32/64 world. *Linux* and *BSD* systems lag behind, while embedded systems for the mobile market such as *Android* and *iOS* are catching up in this area.

While not all packers/obfuscators have been upgraded to handle 64-bit executable formats, there are a lot of tools that can handle both *Windows* PE+ files and ELF 64-bit files. In this tutorial I will describe some of the main differences between the PE and PE+ file formats from the perspective of the binary unpacking process.

PE+ DIFFERENCES

The PE+ file format is a bit like the good old 32-bit *Windows* PE format on steroids. If you thought you would only be able to execute a PE(+) file after successfully booting into *Windows* (you don't have to log in successfully since *Windows* service files are also PE(+) executables internally), you would be wrong. The PE(+) file format is supported by the UEFI specification, so it is possible to execute UEFI PE files even before the target operating system or hypervisor starts. There is one important note: UEFI expects the PE+ file format even on 32-bit architecture, and furthermore it uses just a subset of PE+ features. In turn, the PE+ file format contains a special flag to mark it as UEFI executable.

Other cases for loading Win32 PE or plain PE files are limited today mostly to some DOS-based embedded solutions. But wait a minute – isn't DOS a 16-bit real-mode operating system, whose process loader is limited to handling 64KB COM files and MZ EXEs? How can it execute *Windows* 32-bit protected mode binaries? The answer is simple: DOS extenders.

There are a couple of DOS extenders that offer Win32 PE support out of the box. If you thought that DOS and DOS extenders were part of the past, you would be wrong. Some DOS extenders are still actively being developed and supported: HX DOS Extender [1] is a great example. HX provides a Win32 emulation layer to DOS and enables DOS to load 32-bit PE files.

Returning to our 64-bit version of PE: if you know the PE file format well, you won't be surprised by changes introduced in PE+. The table below summarizes most of the basic ones:

Field	PE	PE+
BaseOfData	ULONG (4 bytes)	Removed from the Optional Header
ImageBase	ULONG (4 bytes)	ULONGLONG (8 bytes)
SizeOfHeapCommit	ULONG (4 bytes)	ULONGLONG (8 bytes)
SizeOfHeapReserve	ULONG (4 bytes)	ULONGLONG (8 bytes)
SizeOfStackReserve	ULONG (4 bytes)	ULONGLONG (8 bytes)
StackOfSizeCommit	ULONG (4 bytes)	ULONGLONG (8 bytes)

Table 1: Comparison between PE and PE+ formats.

The AddressOfEntryPoint field has the same size (ULONG) in both PE and PE+ files. How one can recognize a PE+ file? The magic number field in Optional Headers is different:

Field	PE	PE+
Magic Number	0x10b	0x20b

PE+ executable images are restricted to a maximum size of two gigabytes, so relative addressing with a 32-bit displacement can be used to address static image data. This data includes the import address table, string constants, static global data, and so on.

The rest of the PE+ file looks like a PE file – and what's more important is that all compression/obfuscation tools that handle PE+ files work in exactly the same way as in the case of 32-bit executable images. Therefore, the unpacking process is also similar. The following sections describe some of the other important differences that the 64-bit architecture brings in.

REGISTERS

All general purpose registers are extended to 64-bit width in long mode, providing us with RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP and RIP, which serves like its 32-bit brother EIP as an instruction pointer. New general purpose registers have also been introduced (it seems as if the *AMD* and *Intel* engineers finally decided that they envied some of the good old *Motorola* 68K features): from R8 to R15. New XMM registers are also available: from XMM8 to XMM15. All XMM registers are of 128-bit width. 64-bit MMX0–MMX7 registers are available as well.

CALLING CONVENTION

x64 *Windows* systems no longer use the STDCALL calling convention by default. Instead, the FASTCALL convention is used, which means that the first four parameters are passed in RCX, RDX, R8 and R9 registers. Further parameters are passed using the stack. There are no attempts to spread a single argument across many registers. Additionally, the caller is responsible for allocating parameter space to the callee, and must always allocate sufficient space for the four register parameters, even if the callee doesn't have that many parameters [2].

Following [3], here is a typical function prolog:

```
mov     [RSP + 8], RCX
push    R15
push    R14
push    R13
sub     RSP, fixed-allocation-size
lea     R13, 128[RSP]
```

And here is a typical function epilog:

```
add     RSP, fixed-allocation-size
pop     R13
pop     R14
pop     R15
ret
```

UNAVAILABLE INSTRUCTIONS IN LONG MODE

It is worth mentioning that while in long mode some of the 16/32-bit instructions are unavailable and can generate an undefined opcode exception (#UD).

Furthermore, opcodes from 40h to 4fh (inc register/dec register) have a different mapping in long mode. The REX prefix uses those while in long mode.

String operation instructions like LODSB, STOSB etc. have been extended to handle 64-bit addressing. In turn, a

AAA	POPAD
AAD	POP DS
AAM	POP ES
AAS	POP SS
BOUND	PUSH CS
DAA	PUSH DS
DAS	PUSH ES
INTO	PUSH SS
LDS	PUSHA
LES	PUSHAD
POPA	

Table 2: Unavailable instructions in long mode.

few new string instructions have been introduced: LODSQ, CMPSQ, MOVSQ, SCASQ and STOSQ. As a consequence, REPx prefixes handle 64-bit registers as well as LOOP, LOOPZ and LOOPNZ. All those string instructions can be found in decompression/decryption loops.

Furthermore, both SYSENTER and SYSEXIT instructions are available from legacy mode. In long mode, the SYSCALL/SYSRET pair is used.

If, during unpacking, you see some of these unavailable instructions in your disassembly, you can be assured that either the unpacking process has gone wrong, or it has not yet finished.

WOW64

WOW64 is an emulation layer that enables AMD64 and *Itanium*-based *Windows* systems to execute Win32 applications to maintain backwards compatibility. Figure 1 describes the high-level WOW64 architecture. It is worth mentioning that WoW64.dll loads a 32-bit version of ntdll.dll, which loads other 32-bit DLLs that are needed to support Win32 application execution. Most of these DLLs are exact binary copies from the 32-bit system, however some have been modified in order to be able to share resources with 64-bit system components.

Note that in the case of *Itanium*-based systems there are two more libraries involved in running 32-bit software:

- IA32Exec.bin – contains an x86 software emulator.
- Wowia32x.dll – provides an interface between WOW64 and IA32Exec.bin.

THE TEST FILES

Since this is a tutorial, I've decided not to use a specific malware sample. Instead, I have created a sample PE+ file

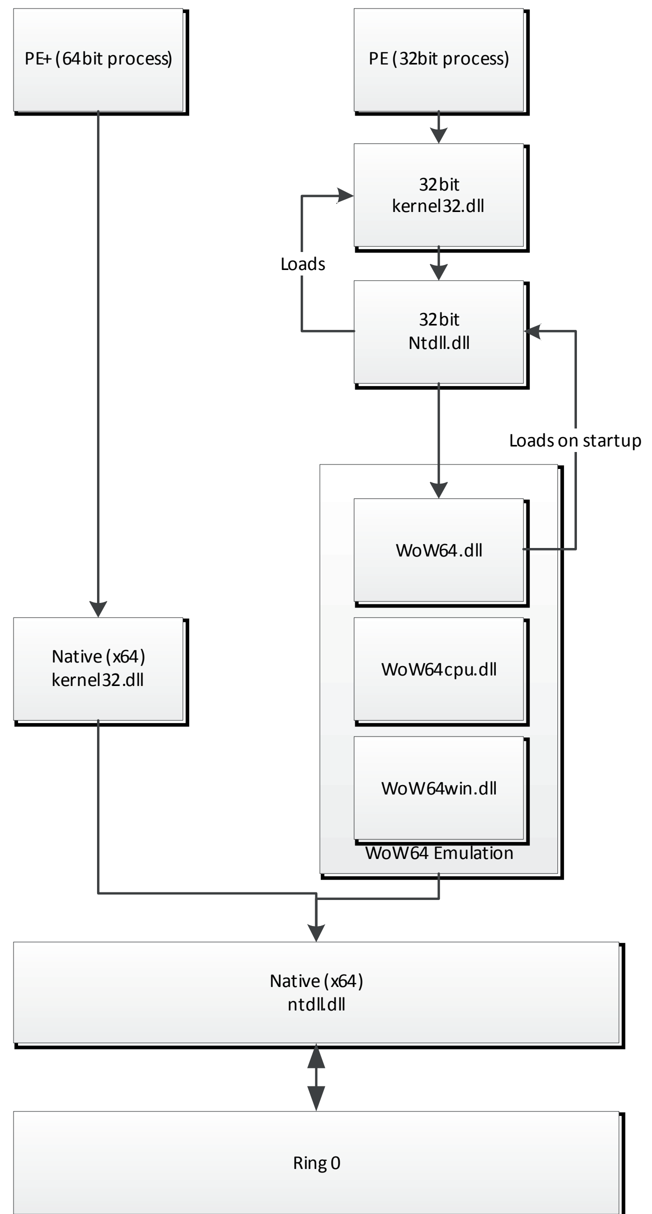


Figure 1: WOW64 architecture.

written in assembly language. This file can be compiled with flat assembler (fasm), which is available at [4]. Do not try to compile this example with different assemblers such as MASM or NASM as you will not succeed without editing the source code. The presented examples use specific fasm syntax. I've chosen fasm since it provides a lot of control over output executable files within the source code level and no external linker is needed in our case. For example, you can manually control the layout of PE+ sections, their order and attributes:

```
; Example of 64-bit PE program

format PE64 GUI
entry start     ;Entry point definition

;DATA SECTION
section '.data' data readable writeable

   _caption db 'Win64 assembly program',0
   _message db 'Hello World!',0

;CODE SECTION
section '.text' code readable executable

   start:
        sub     rsp,8*5  ; reserve stack for API use
and make stack dqword aligned

        mov     r9d,0
        mov r8,_caption
        mov rdx,_message
        xor rcx,rcx
        call    [MessageBoxA]

        mov     ecx,eax
        call    [ExitProcess]

; IMPORT SECTION
section '.idata' import data readable writeable

   dd 0,0,0,RVA krnl_name,RVA krnl_tbl
   dd 0,0,0,RVA user_name,RVA user_tbl
   dd 0,0,0,0,0

   krnl_tbl:
     ExitProcess dq RVA _ExitProcess
     dq 0
   user_tbl:
     MessageBoxA dq RVA _MessageBoxA
     dq 0

   krnl_name db 'KERNEL32.DLL',0
   user_name db 'USER32.DLL',0

   _ExitProcess dw 0
     db 'ExitProcess',0
   _MessageBoxA dw 0
     db 'MessageBoxA',0
```

To compile the file just enter: fasm.exe testwin64.asm. Assuming that the compilation succeeded you can now load the binary file into *IDA Pro* using the standard Open File option. This will be our template file that we will use for all further operations. The file sections and attributes are shown in Figure 2.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
.data	0000000000401000	0000000000402000	R	W	.	.	L	para	0001	public	DATA	64
.text	0000000000402000	0000000000403000	R	.	X	.	L	para	0002	public	CODE	64
.idata	0000000000403000	0000000000404000	R	W	.	.	L	para	0003	public	DATA	64

Figure 2: Section list of test file before compression.

Next, disassemble the entry point using the Ctrl+E shortcut to jump directly to the start label, as shown in Figure 3. You can see that the data closely resembles our fasm source – now you know why I have chosen fasm for this job: the source code is quite similar to the resulting EXE file.

```
.text:0000000000402000                         ; =============== S U B R O U T I N E ===============
.text:0000000000402000
.text:0000000000402000                         ; Attributes: noreturn
.text:0000000000402000
.text:0000000000402000                                 public start
.text:0000000000402000                         start   proc near
.text:0000000000402000 48 83 EC 28                     sub     rsp, 28h
.text:0000000000402004 41 B9 00 00 00 00               mov     r9d, 0                  ; uType
.text:000000000040200A 49 C7 C0 00 10 40+              mov     r8, offset Caption      ; lpCaption
.text:0000000000402011 48 C7 C2 17 10 40+              mov     rdx, offset Text        ; lpText
.text:0000000000402018 48 31 C9                        xor     rcx, rcx                ; hWnd
.text:000000000040201B FF 15 2B 10 00 00               call    cs:MessageBoxA
.text:0000000000402021 89 C1                           mov     ecx, eax                ; uExitCode
.text:0000000000402023 FF 15 13 10 00 00               call    cs:ExitProcess
.text:0000000000402023                         start   endp
.text:0000000000402023
```

Figure 3: Entry point and main code of the test file.

Take a note of the instruction bytecodes forming the entry point and entry point address: 0x0402000. This address will later be our original entry point address (OEP).

Next, let's inspect the import section and list imports using the 'Imports' subview from *IDA Pro* (Figure 4). Since we have used only two functions, MessageBoxA and ExitProcess, only those two are listed.

Address	Ordinal	Name	Library
000000000...		ExitProcess	KERNEL32
000000000...		MessageBoxA	USER32

Figure 4: Test file imports.

The next step is to generate the target file. In order to do that we will compress our test file so that we will be able to make a comparison with the original during the unpacking process.

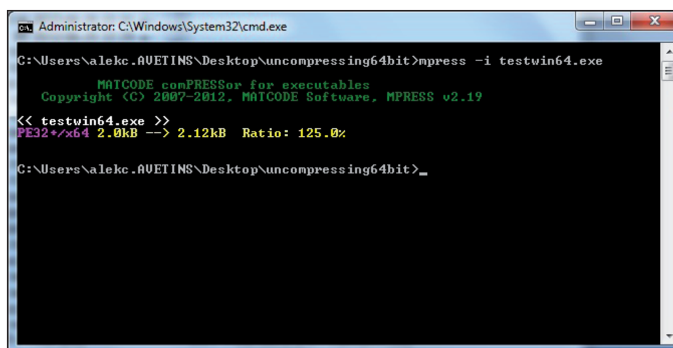


Figure 5: Generating a compressed file using mpress.

I've chosen the mpress [5] file packer since it is freely available and handles both PE and PE+ files. In order to create a new, compressed executable file, follow the commands shown in Figure 5. We use –i options since the resulting compressed file will be larger than the original

one. By default, mpress refuses the compression and creation of a new executable if the resulting output file is bigger than the input. Observant readers might notice that our test file can also be used as a base for measuring the efficiency of compression algorithms. Additionally, the test file is a perfect target for reversing the compression stub since the original EXE file has such a simple construction.

IDA PRO NATIVE DEBUGGER VS IDA PRO BOCHS PLUG-IN

Obviously, any native 64-bit debugger supported by *IDA Pro* requires *Windows* on the x64 platform. Fortunately, the *Bochs* plug-in allows you to debug both PE and PE+ binaries inside *Bochs*, even on 32-bit platforms. The speed impact due to code emulation can be ignored in most cases during malware analysis and unpacking files. The next advantage of the *Bochs* plug-in, when analysing hostile code, is that the code is 'executed' in a virtual, controlled environment. The recently disclosed SYSRET privilege escalation vulnerability (CVE-2012-0217) demonstrates the risk associated with running hostile code inside hypervisors. The disadvantage of emulation is obvious – there are no 100% perfect emulators of bare metal hardware and the real operating system. There is a set of methods that can be used to detect if code is being executed under *Bochs* emulation. For some of the most basic methods see [7].

When using *Bochs* in PE operation mode, keep in mind that in the current version there are some important limitations:

- PE+ support is limited.
- *Windows* environment emulation is limited and this can lead to its easy detection by the process.
- Thread and process manipulations are not supported – this could render the *Bochs* plug-in useless against more advanced compression/obfuscation methods combined with anti-debugging tricks.
- Only a handful of API calls are implemented.
- LoadLibrary() works only on DLLs defined in the startup.idc file before running the debugger.

Fortunately, some important *Windows* features such as TLS callbacks, SEH and crucial *Windows* structures are available. Furthermore, bochsys.dll exports the BxUndefinedApiCall() function, which catches unimplemented API calls. Setting a breakpoint on it allows such a situation to be trapped or for the end of the unpacking process to be detected. Bochsys.dll exports another useful function: BxIDACall(). Setting a breakpoint on this function allows all API calls that are handled internally by *IDA Pro* to be monitored.

UUNP PLUG-IN

The uunp plug-in is a demonstration plug-in bundled with *IDA Pro*. It is available from the 'Edit->Plugins-> Universal unpacker manual reconstruct' menu option. As a side note: *Windows* 32-bit plug-ins use the *.plw file extension, while 64-bit ones use *.p64. They all reside in the plug-ins directory of the *IDA Pro* installation folder. Looking at the limitations of the *Bochs* plug-in and some additional information required by the uunp plug-in (Figure 6), you might be wondering why we are not using another plug-in distributed with *IDA Pro*: Universal PE Unpacker. We will discuss the Universal PE Unpacker internals in the second part of this tutorial.

The uunp plug-in does the following:

1. Locates the Import Address Table (IAT).
2. Creates an XTRN segment to represent imports.
3. Generates a new entry point (OEP) in the IDA database while deleting the old one used by the packer.
4. Forces reanalysis of new code sections.

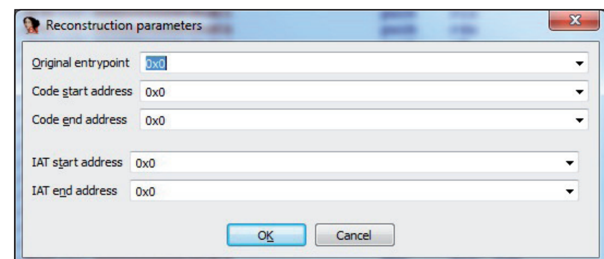


Figure 6: uunp plug-in main window – you need to enter the correct information manually in order to get the desired results.

However, in order to get a reasonable output from the uunp plug-in we need to feed it manually with the proper addresses of the original file. The only way we can find out the requested information is to execute or emulate decompression code. The most important pieces of information we need to gather are: the original entry point (OEP) address and the Import Address Table (IAT) start and end address. The value for the 'Code end address' field could theoretically be guessed, however this is not recommended when analysing malware.

UNPACKING PE+ WITH IDA, BOCHS AND UUNP

Let's start with the default PE+ file loader from *IDA Pro* – in order to do that, just open the compressed test file. The

default PE+ file loader (Figure 7) will warn us about the Import Table section (Figure 8).

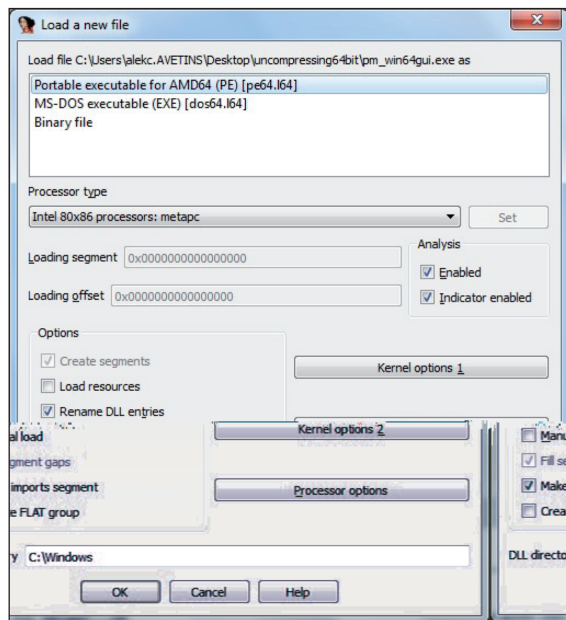


Figure 7: Loading the compressed file – note that the 'Make imports segment' option is enabled by default.

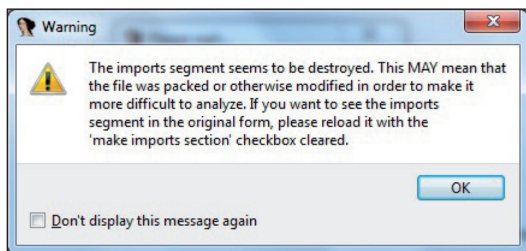


Figure 8: IDA Pro-generated warning during the loading of a PE+ file with a strange Import Table.

Next we should examine our PE+ file layout in memory using the option 'View->Open subviews->Segments' (shift+F7 is the default shortcut). Figure 9 shows that there are three segments, named .MPRESS1, .MPRESS2, .MPRESS2 and .idata (this is not a mistake: the .MPRESS2 name is used twice, but the two segments have different start addresses). Note that segments in *IDA Pro* are not directly equal to executable file sections. In our case, segments have been created automatically by *IDA Pro*. A different list of segments will be created if we load our PE+ file with the 'Make imports segment' option disabled.

Since the name '.idata' suggests that *IDA Pro* has somehow created an Import Address Table section (marked as XTRN), we can inspect it, but first let's check which imports *IDA* detected. Use the 'View->Open subviews->Imports' option to list all imports (Figure 10). Only three *Windows* functions are imported: GetModuleHandleA, GetProcAddress and MessageBoxA. Inspection of the '.idata' segment confirms our findings (Figure 11). At least one obvious function import is missing from this picture: LoadLibrary and VirtualProtect come to mind.

Since it was detected in the imports, we can assume that GetProcAddress is being used by the decompression loop. Therefore, we can either manually analyse and trace code under the debugger in order to find its invocation or we can set up a breakpoint at GetProcAddress. Since this is a tutorial, setting up a breakpoint at GetProcAddress is not a bad idea. It will not only allow us to verify our hypothesis that functions found in the import table are used to recreate the original IAT, but also to inspect how *IDA Pro* cooperates with *Bochs* at a low level. This knowledge may be helpful in the future in case of more advanced assignments.

Before running the *Bochs* debugger plug-in we need to configure it. From the 'Debugger' menu choose 'Select debugger option'. From this window select 'Local Bochs debugger' (see Figure 12).

Next, again from the 'Debugger' menu, select 'Debugger options…' – a new configuration window will open (Figure 13). From this window click the 'Set specific options' button to display another window, as shown in Figure 14.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
.MPRESS1	0000000000401000	0000000000404000	R	W	X	.	L	para	0001	public	CODE	64
.MPRESS2	0000000000404000	000000000040403C	R	W	X	.	L	para	0002	public	CODE	64
.idata	000000000040403C	000000000040405C	R	W	X	.	L	para	0002	public	XTRN	64
.MPRESS2	000000000040405C	0000000000405000	R	W	X	.	L	para	0002	public	CODE	64

Figure 9: IDA Pro automatically generates segments of the compressed file with the 'make imports segment' options enabled.

Address	Ordinal	Name	Library
000000000...		GetModuleHandleA	KERNEL32
000000000...		GetProcAddress	KERNEL32
000000000...		MessageBoxA	USER32

Figure 10: Compressed file imported functions – LoadLibrary is missing, for example.

```
.idata:000000000040403C ; Imports from KERNEL32
.idata:000000000040403C ;
.idata:000000000040403C ; ===========================================================================
.idata:000000000040403C
.idata:000000000040403C ; Segment type: Externs
.idata:000000000040403C ; _idata
.idata:000000000040403C ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
.idata:000000000040403C                 extrn GetModuleHandleA:qword
.idata:000000000040403C                                         ; DATA XREF: .MPRESS2:__IMPORT_DESCRIPTOR_KERNEL32↑o
.idata:000000000040403C                                         ; .MPRESS2:0000000000404010↑o
.idata:0000000000404044 ; FARPROC __stdcall GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
.idata:0000000000404044                 extrn GetProcAddress:qword
.idata:000000000040404C
.idata:0000000000404054 ;
.idata:0000000000404054 ; Imports from USER32.DLL
.idata:0000000000404054 ;
.idata:0000000000404054 ; int __stdcall MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
.idata:0000000000404054                 extrn MessageBoxA:qword ; DATA XREF: .MPRESS2:__IMPORT_DESCRIPTOR_USER32↑o
.idata:0000000000404054                                         ; .MPRESS2:0000000000404024↑o
.idata:0000000000404054
```

Figure 11: Inspection of the .idata segment.

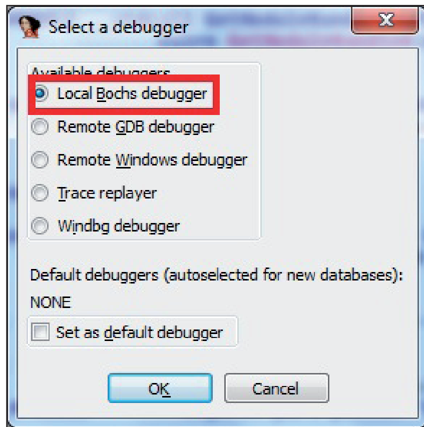


Figure 12: Selecting Bochs local debugger as default for this session.

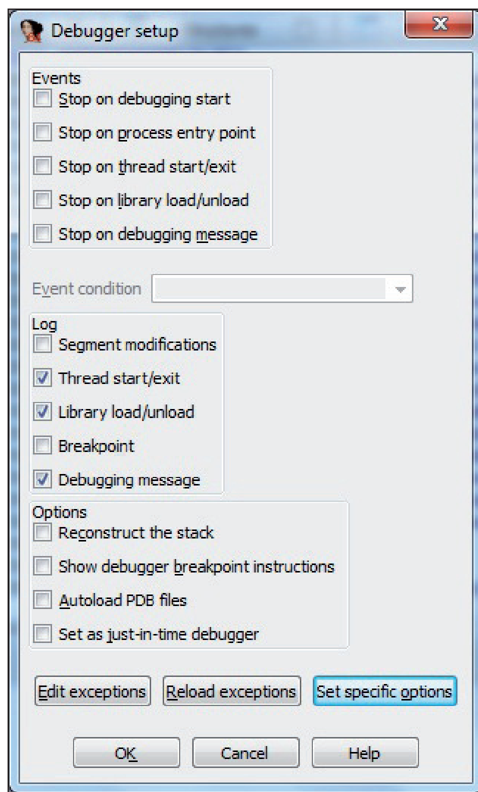


Figure 13: Configuring basic debugger options.

Be sure to enable 64-bit support and PE file support in this window.

Now we are ready to start unpacking our target file. First go to the PE+ entry point – this can be done by pressing Ctrl+E and selecting one of the possible

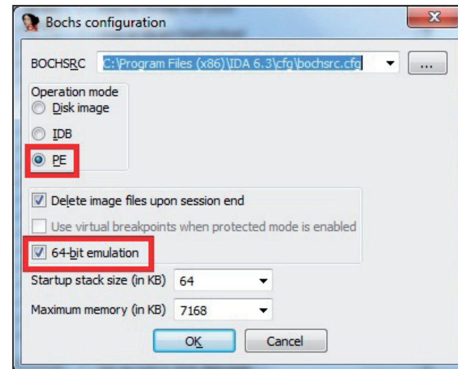


Figure 14: Bochs specific options.

addresses (Figure 15). In our case, *IDA Pro* detected only one entry point and labelled it 'start'. This is obviously not our Original Entry Point. Let's add a breakpoint at the entry point. Press F2 at the entry point (0x04040C2 address in our case) and start a debugger. This can be done either by pressing the green 'play' icon on the toolbar or by pressing the F9 key. Take a second to look at the navigation bar – the current entry point is located near the end of the file: many compression/obfuscation tools just add their code after the original file end. This could be a hint that the OEP may be located below the current entry point, however at this point this is only a hypothesis.

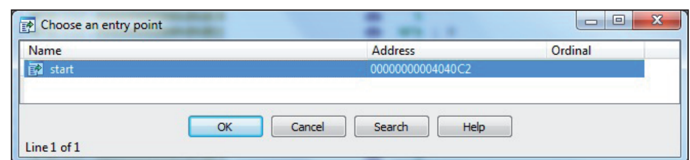


Figure 15: Selecting the entry point.

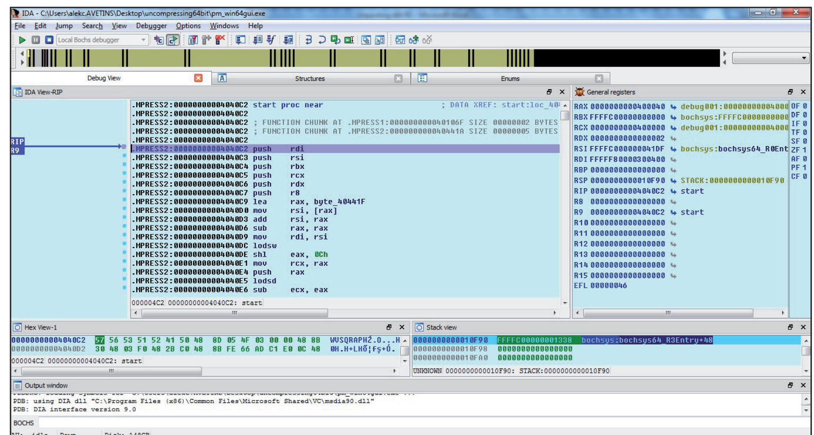


Figure 16: Breakpoint hit at the entry point.

After pressing the F9 key (Run), the debugger should stop at the first instruction. Now we can finally add a breakpoint at the GetProcAddress function. In order to do so, open the ‘Breakpoints’ list from the ‘Debugger->Breakpoints->Breakpoint list’ menu. Now press the ‘insert’ key to add a new breakpoint. At the ‘location’ field enter ‘kernel32_GetProcAddress’ (remember that the kernel32 name is misleading since we are dealing with the 64-bit version despite the ‘32’ in the name) and click ‘OK’. Now, run the debugger again (F9) and wait until the GetProcAddress function breakpoint has been hit. Our function should look like this:

```
KERNEL32.dll:0000000078D26455 kernel32_
GetProcAddress:
KERNEL32.dll:0000000078D26455 push   cs:off_78D2645C
KERNEL32.dll:0000000078D2645B retn
```

Use ‘step into’ (F7) options to execute the retn instruction. The next function should be within the bochsys module:

```
bochsys:FFFFC00000001467 bochsys64_BxGetProcAddress:
; DATA XREF: KERNEL32.dll:off_78D2645Co
bochsys:FFFFC00000001467 mov         rax,
0FFFFC00000001467h
bochsys:FFFFC00000001471 call        near ptr
bochsys64_BxIDACall
bochsys:FFFFC00000001476 retn
```

We can ‘step over’ this code until we reach the retn instruction. This is a stub code used by the *Bochs* plug-in to communicate with *IDA*, as mentioned earlier. After executing the retn instruction we return to our module inside the .MPRESS1 section:

```
.MPRESS1:00000000004010D4 test   eax, eax
.MPRESS1:00000000004010D6 jz     short loc_401103
.MPRESS1:00000000004010D8 push   rax
.MPRESS1:00000000004010D9 push   rsp
.MPRESS1:00000000004010DA pop    r9
```

This is obviously the code that checks the success of GetProcAddress (test eax,eax). Now let’s open the Imports window and jump to GetProcAddress import (Figure 17):

Now you see there is a cross reference j_GetProcAddress – jump to it (Figure 18).

There is another cross reference at .MPRESS1:0x0401152. Once again, jump to that cross reference to find the following code:

```
MPRESS1:000000000040114F loc_40114F:
        ; CODE XREF: .MPRESS1:0000000000401141j
.MPRESS1:000000000040114F mov    rcx, rbx
        ; hModule
.MPRESS1:0000000000401152 call   j_GetProcAddress
.MPRESS1:0000000000401157 stosq
.MPRESS1:0000000000401159
.MPRESS1:0000000000401159 loc_401159:
        ; CODE XREF: .MPRESS1:0000000000401161j
.MPRESS1:0000000000401159 xor    al, al
.MPRESS1:000000000040115B mov    [rsi-1], al
.MPRESS1:000000000040115E lodsb
.MPRESS1:000000000040115F or     al, al
.MPRESS1:0000000000401161 jnz    short loc_401159
.MPRESS1:0000000000401163 jmp    short loc_401132
```

The stosq instruction should store the address returned by the GetProcAddress() function at the location pointed to by the RDI register. The RDI register value during the first iteration of this loop will point to the original IAT. Consequently, at this address the RDI register during the last iteration will point to the end of the IAT. Note both values, since these are required by the uunp plug-in.

Stepping over this loop we can see how the IAT is being reconstructed and finally, when we reach the following code, we have found the jump to the original entry point:

```
MPRESS1:0000000000401165 exit_to_oep:
        ; CODE XREF: .MPRESS1:0000000000401118j
.MPRESS1:0000000000401165 lea    rdi, loc_40106F
.MPRESS1:000000000040116C mov    al, 0E9h
.MPRESS1:000000000040116E stosb
.MPRESS1:000000000040116F mov    eax, 10Ch
.MPRESS1:0000000000401174 stosd
.MPRESS1:0000000000401175 add    rsp, 28h
```

```
MPRESS2:000000000040403C                                   ; .MPRESS2:__IMPORT_DESCRIPTOR_KERNEL32↑
MPRESS2:0000000000404044 ; FARPROC __stdcall GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
MPRESS2:0000000000404044 GetProcAddress dq offset kernel32_GetProcAddress
MPRESS2:0000000000404044                                   ; DATA XREF: j_GetProcAddress↑r
```

Figure 17: GetProcAddress import.

```
MPRESS1:0000000000401193 ; FARPROC __stdcall j_GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
MPRESS1:0000000000401193 j_GetProcAddress proc near        ; CODE XREF: .MPRESS1:0000000000401152↑p
MPRESS1:0000000000401193 jmp     cs:GetProcAddress
MPRESS1:0000000000401193 j_GetProcAddress endp
```

Figure 18: GetProcAddress jump.

```
.MPRESS1:0000000000401179 pop    r8
.MPRESS1:000000000040117B pop    rdx
.MPRESS1:000000000040117C pop    rcx
.MPRESS1:000000000040117D pop    rbx
.MPRESS1:000000000040117E pop    rsi
.MPRESS1:000000000040117F pop    rdi
.MPRESS1:0000000000401180 jmp    OEP_at_0x402000
```

A few important observations should be made at this point:

- The packer does not use the popa instruction before jumping to the original entry point (some 32-bit compressors use it). Therefore, any universal unpacking methods based on detecting the popa instruction before jumping to OEP will fail. Popa/popad is not valid in long mode, as mentioned earlier (however POPFQ is).
- We can use the long list of pop instructions ending with the jmp as a signature to look for the original entry point address. Note that our OEP is actually at a higher address than the decompression exit code. This means that any plug-in trying to automatically detect the OEP based on a jump below the decompression loop in memory will also fail.

At this point we can feed the uunp plug-in with the data we have gathered during our debugging session.

IDA PRO ALTERNATIVE STRATEGIES

Manual unpacking obviously does not scale well in production environments. Therefore, plug-ins like uunp can be treated only as a simple demonstration of *IDA Pro*'s scripting abilities and plug-ins/modules architecture. If you are willing to automate the unpacking process with *IDA Pro*, or the case you are working on requires some special treatment/tricks, you have a couple of options that might help you:

- Write a custom loader module – all examples here were based on *IDA Pro* default PE+ loader. However, you can either load a file manually, bypassing the loader (this option is quite handy when some uncommon PE+ format tricks are used), or implement your own loader. This could be handy if you are able to automatically decompress original code and data plus reconstruct the import table. Obviously this requires either some knowledge about how a certain packer works, or use of a more generic approach based on execution/emulation of code.
- Write a custom processor module – this option is especially handy when, besides the compression/encryption algorithm, some kind of virtual machine/bytecode scheme has been used in order to further obfuscate the original executable code.

FINAL NOTES AND CHALLENGES

It turns out that unpacking 64-bit PE files doesn't really differ much from unpacking 32-bit EXEs or DLLs. The only difference is the limited number of tools that can handle the PE+ format correctly.

Furthermore, both 32- and 64-bit architectures allow complex compression, encryption and obfuscation techniques, and since PE(+) structures add some complexity to the equation, we are yet to see new techniques. Of course, the complexity of PE+ will increase as natural *Windows* platform evolution introduces new bugs and vulnerabilities into the process loader. I'm afraid that those vulnerabilities are likely to be exploited sooner rather than later.

In the second part of this tutorial (which will appear in the August issue of *VB*) I will dig a bit more deeply into *Windows* x64 internals, use some of *IDA Pro*'s scripting functionality and use *WinDbg* to unpack our example file. In the meantime, if you would like to see another example of unpacking an mpress binary with *IDA Pro* take a look at the blog post at [9].

REFERENCES

[1] HX DOS Extender. http://www.japheth.de/HX.html.

[2] MSDN: Overview of x64 Calling Conventions. http://msdn.microsoft.com/en-us/library/ms235286(v=vs.80).aspx.

[3] MSDN: Prolog and Epilog. http://msdn.microsoft.com/en-us/library/tawsa7cb(v=vs.80).aspx.

[4] flat assembler. http://flatassembler.net/.

[5] mpress. http://www.matcode.com/mpress.htm.

[6] UPX. http://upx.sourceforge.net/.

[7] Ferrie, P. Attacks on Virtual Machine Emulators. https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.

[8] WinDbg. http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx.

[9] Unpacking mpress'ed PE+ DLLs with the Bochs plugin. http://www.hexblog.com/?p=403.

TUTORIAL 2

QUICK REFERENCE FOR MANUAL UNPACKING II

Abhishek Singh
FireEye, USA

Unpacking is a critical step in the process of analysing malware. Malware authors use packers to make it difficult for their malware to be reversed – the packers encode the original instructions. By packing a malicious executable, the author can be sure that when it is opened in a disassembler it will not show the correct sequence of instructions. Packers add some instructions at the top of the binary to unpack the executable.

In [1] we described the steps that can be used to manually unpack malware packed with a number of commonly used packers. In this article, we will cover another set of packers that are popular with malware authors.

(The purpose of this article is to provide a quick reference guide that will assist analysts in the unpacking of malware and reduce the response time for malware analysis – the full technical details of each packer have thus been omitted.)

Molebox

Molebox is a runtime executable packer for *Windows* applications. It can be used to pack an application and all its data into a single executable file. As shown in Figure 1, Molebox starts with a CALL instruction, followed by PUSHAD.

The first step in the process of locating the original entry point (OEP) of a file packed with Molebox is to put a breakpoint on PUSHAD. PUSHAD pushes the contents of the general purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI and EDI (if the current operand-size attribute is 32), AX, CX, DX, BX, SP (original value), BP, SI and DI (if the operand size attribute is 16).

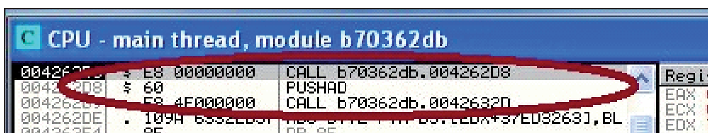


Figure 1: The initial instructions for Molebox.

Step once after the breakpoint has triggered, since EDI is the last value which is pushed (as shown in Figure 2). The next step involves setting an access hardware breakpoint at the memory location pointed to by ESP. As shown in Figure 2, the address location at ESP stores the value in EDI.

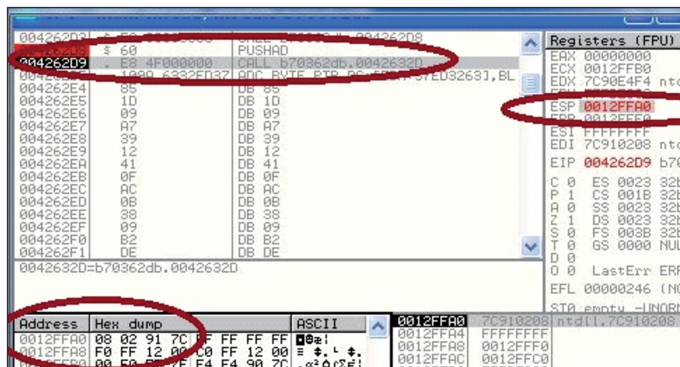


Figure 2: EDI in the memory location pointed to by ESP.

When the hardware breakpoint triggers, a POP EAX instruction is followed by the CALL EAX instruction (see Figure 3). Step into CALL EAX, and we have reached the OEP.

Simply dump the process to obtain the unpacked file.

Figure 3: The instructions when the breakpoint triggers.

PE-Pack

PE-Pack was released by ANAKiN. It is commonly used by malware authors to hide code. When a packed file is launched in a debugger, it starts with a JMP instruction (see Figure 4).

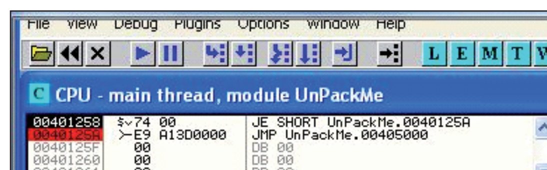


Figure 4: The initial instruction for PE-Pack.

To unpack the file, put a breakpoint on the JMP instruction. When the breakpoint triggers, the debugger will reach the PUSHAD instruction, as shown in Figure 5.

The PUSHAD instruction will push all the registers onto the stack. Next, set an access hardware breakpoint on the uppermost dword of the stack. As shown in Figure 5, the top dword will be the same value as that stored in the EDI register. This can be confirmed by looking in memory at the address value pointed to by ESP.

When the hardware breakpoint triggers, as shown in Figure 6, the debugger will reach the instruction JMP EAX. This is the jump to the OEP. Step once on it and we have reached the OEP.

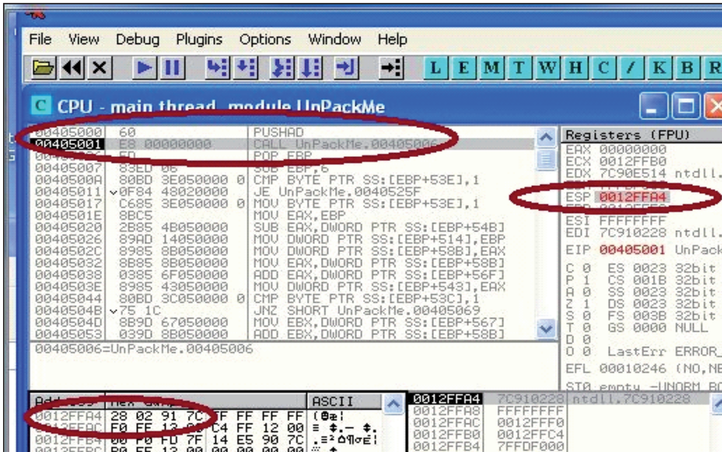


Figure 5: The hardware breakpoint when PUSHAD is encountered.

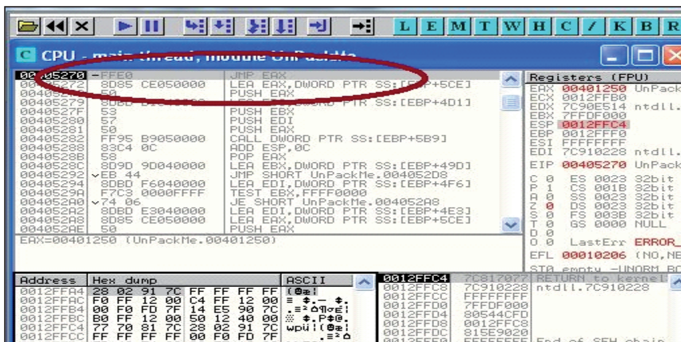


Figure 6: Instructions encountered when the debugger reaches the OEP.

WinUpack

WinUpack is a command-line program for compressing/ decompressing *Windows* EXE/DLL files. The program is mainly used for compression, rather than for protection. When the file is executed, the compressed EXE/DLL will decompress and run normally without any additional software. In order to unpack a WinUpack packed file, it should be loaded in *OllyDbg*. Once the file has been loaded in the debugger, scroll down to find the following set of instructions:

```
46                  inc esi
AD                  lodsd
85C0                test eax, eax
0F84xxxxxxxx 00     jz xxxxxxxx
```

When JZ triggers, we have reached the OEP. Dump the process to get the unpacked file.

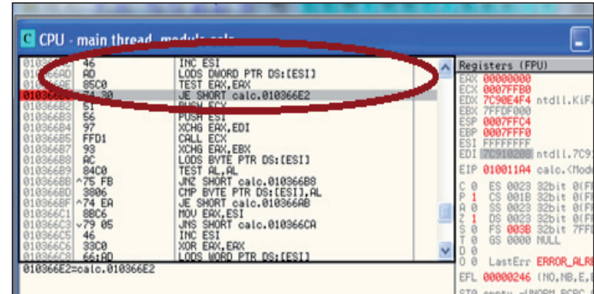


Figure 7: Assembly instructions just before the OEP.

PolyCryptPE

PolyCryptPE is used for encrypting PE files and can also be used to protect PE files from disassembly. The packer starts with the PUSHAD instruction. However, as seen in Figure 8, it employs various anti-debugging tricks.

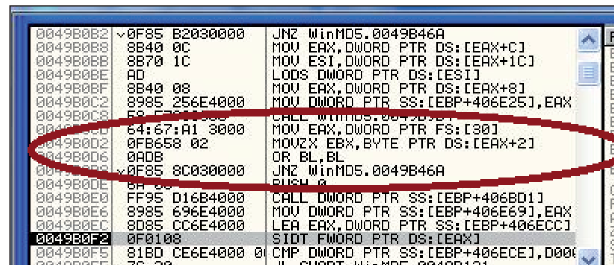


Figure 8: Assembly instructions checking for a debugger.

The instruction MOV EAX, DWORD PTR FS:[30] loads the value FS:[30] (or process environment block) in the EAX register. The next instruction, MOVZX EBX, BYTE PTR DS:[EAX+2], loads the third value of PEB. When the process is being debugged, the third value in the PEB structure will be set. So when a file packed with PolyCryptPE is being debugged, either the isDebugged() flag must be set, or the HideOE plug-in must be used in order to hide the debugger.

The packed file starts with the POPAD instruction. In order to debug the packed file, the POP EBP, POPFD, POPAD instructions must be located. When the debugger executes the RETN instruction, the debugged process has reached the OEP. The process should be dumped to get the unpacked version of the file.

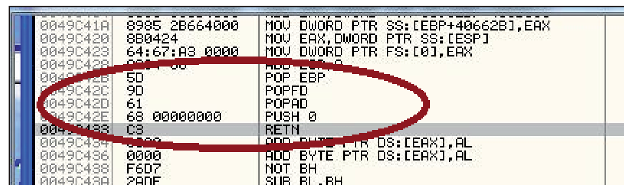


Figure 9: The assembly instructions before the OEP.

SimplePack

SimplePack is another packer that is often used by malware authors. It uses LZMA compression. When the packed process is opened in a debugger, the packed code starts with the PUSHAD instruction, as shown in Figure 10. The instruction will push all the general purpose registers onto the stack, with the value stored in EDI being on top of the stack.

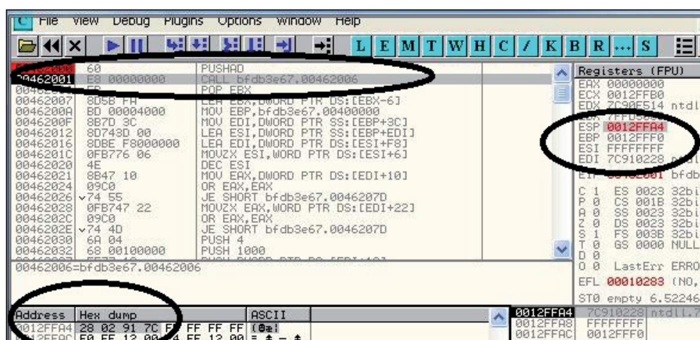


Figure 10: Starting instructions of the packed file.

The next obvious step in unpacking the file is to set a hardware access breakpoint on the uppermost dword of the stack when the PUSHAD instruction is executed. When the breakpoint triggers, as shown in Figure 11, the POPAD, PUSH and RETN instructions are encountered.

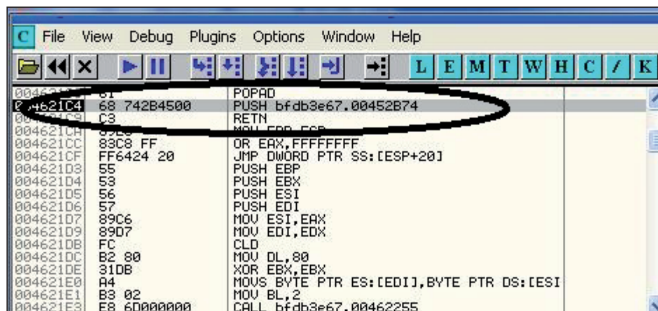


Figure 11: Assembly instructions when the hardware breakpoint triggers.

When the debugger executes the RETN instruction, we can see the initialization of the stack frame (Figure 12).

Upon initialization of the stack frame, the debugged process can be dumped to get the unpacked file.

PECompact 1.x

PECompact 1.x is a commercial packer. It compresses the code, date, import directory, selected resources and other portions of *Windows* executables (DLL, EXE, SCR, OCX etc.). The concept for unpacking PECompact 1.x is pretty

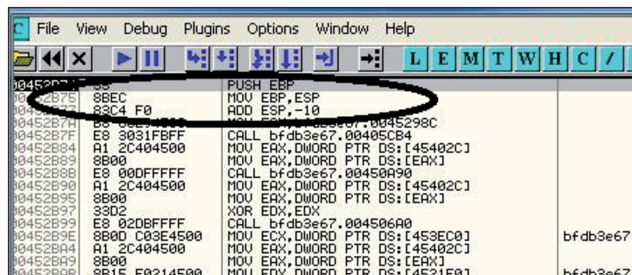


Figure 12: SimplePack initialization of the stack frame.

much the same as that described for SimplePack above. After loading the packed file in a debugger, step down a few instructions and find PUSHAD. The instruction will push the registers onto the stack. When the instructions have been pushed, set a hardware access breakpoint on the top dword of the stack. Basically, the hardware access breakpoint will be on the ESI value at the address pointed to by ESP. When the breakpoint is triggered, the following instructions will be encountered:

```
9D              popfd
50              push
68 xx xx xx xx  push xxxxxxxx
c2 04 00        retn 04
```

RETN 04 is a jump to OEP. Step on the instruction and then dump the process to get the unpacked file.

CONCLUSION

Unpacking is a key step for the static analysis of malware. Loading a packed malicious executable and executing step-by-step instructions in a debugger is one of the best ways to locate the OEP. In this article, we have provided assembly instructions for Molebox, PE-Pack, WinUpack, PolyCryptPE, PECompact 1.x and SimplePack – these instructions can be used to manually unpack malware. It is also possible to generate OllyScript for the methods mentioned in the article. Open RCE [2] provides a good reference collection of OllyScripts for unpacking these packers.

REFERENCES

[1] Singh, A. Quick reference for manual unpacking. Virus Bulletin, April 2012, p.11. http://www.virusbtn.com/virusbulletin/archive/2012/04/vb201204-manual-unpacking.

[2] OllyScripts for the Packers. http://www.openrce.org/downloads/browse/OllyDbg_OllyScripts.

END NOTES & NEWS

Black Hat USA will take place 21–26 July 2012 in Las Vegas, NV, USA, followed immediately by DEFCON 20, which takes place 26–29 July. For more information see http://www.blackhat.com/ and https://www.defcon.org/.

The 21st USENIX Security Symposium will be held 8–10 August 2012 in Bellevue, WA, USA. For more information see http://usenix.org/events/.

TakeDownCon Baltimore is scheduled to take place 25–30 August 2012 in Baltimore, MD, USA. Interest can be registered at http://www.takedowncon.com/Events/Baltimore.aspx.

SOURCE Seattle 2012 takes place 13–14 September 2012 in Seattle, WA, USA. A call for papers has been announced, with a deadline date of 25 June. For more information see http://www.sourceconference.com/seattle/.

VB2012 will take place 26–28 September 2012 in Dallas, TX, USA. Online registration is now available. Full details can be found at http://www.virusbtn.com/conference/vb2012/.

Security Summit Verona takes place 4 October 2012 in Verona, Italy. For details see https://www.securitysummit.it/.

RSA Conference Europe takes place 9–11 October 2012 in London, UK. Registration is now open. http://www.rsaconference.com/events/2012/europe/.

Ruxcon takes place 20–21 October 2012 in Melbourne, Australia. A call for papers has been announced, with a deadline date of 15 July. See http://www.ruxcon.org.au/.

eCrime 2012 will be held 22–25 October 2012 in Las Croabas, Puerto Rico, consisting of the APWG annual General Members Meeting and the eCrime Researchers Summit VII. During the General Meeting the APWG will examine crimeware's evolution, behavioural vulnerabilities and human factors that contribute to its success, the roles of registrars, registries and DNS in managing phishing attacks, public health approaches to managing the ecrime, as well as news on counter-ecrime efforts and resources. The eCrime Researchers Summit will discuss all aspects of electronic crime and ways to combat it. For details see http://apwg.org/events/events.html.

Hacker Halted USA will take place 25–31 October 2012 in Miami, FL, USA. http://www.hackerhalted.com/.

AVAR 2012 will be held 12–14 November 2012 in Hang Zhou, China. For details see http://www.aavar.org/avar2012/.

SOURCE Barcelona 2012 takes place 16–17 November 2012 in Barcelona, Spain. For details see http://www.sourceconference.com/barcelona/.

TakeDownCon Las Vegas is scheduled to take place 1–6 December 2012 in Las Vegas, NV, USA. Interest can be registered at http://www.takedowncon.com/Events/LasVegas.aspx.

VB2013 will take place 2–4 October 2013 in Berlin, Germany. Details will be revealed in due course at http://www.virusbtn.com/conference/vb2013/. In the meantime, please address any queries to conference@virusbtn.com.

ADVISORY BOARD

Pavel Baudis, *Alwil Software, Czech Republic*
Dr Sarah Gordon, *Independent research scientist, USA*
Dr John Graham-Cumming, *CloudFlare, UK*
Shimon Gruper, *NovaSpark, Israel*
Dmitry Gryaznov, *McAfee, USA*
Joe Hartmann, *Microsoft, USA*
Dr Jan Hruska, *Sophos, UK*
Jeannette Jarvis, *McAfee, USA*
Jakub Kaminski, *Microsoft, Australia*
Eugene Kaspersky, *Kaspersky Lab, Russia*
Jimmy Kuo, *Microsoft, USA*
Chris Lewis, *Spamhaus Technology, Canada*
Costin Raiu, *Kaspersky Lab, Romania*
Péter Ször, *McAfee, USA*
Roger Thompson, *Independent researcher, USA*
Joseph Wells, *Independent research scientist, USA*

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- *Bona fide* charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):

- Comparative subscription: $100

See http://www.virusbtn.com/virusbulletin/subscriptions/ for subscription terms and conditions.

Editorial enquiries, subscription enquiries, orders and payments:

Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

This publication has been registered with the Copyright Clearance Centre Ltd. Consent is given for copying of articles for personal or internal use, or for personal use of specific clients. The consent is given on the condition that the copier pays through the Centre the per-copy fee stated below.

VIRUS BULLETIN © 2012 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England. Tel: +44 (0)1235 555139. /2012/$0.00+2.50. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the publishers.