MALWARE DEVELOPMENT LIFE CYCLE

Raymond Roberts
Microsoft, Level 5, 4 Freshwater Place, Southbank, Victoria 3006, Australia

Email raymond.roberts@microsoft.com

ABSTRACT

In recent years malware has transcended its 'not so humble' beginnings to evolve in complexity to rival many decent sized software projects. This reflects the increased sophistication of the producers of the malicious content and is reflected in their methods of attack.

This indicates an improvement in development methodologies that enables malware producers to improve their output, adding functionality and capabilities to achieve maximum gain.

In this paper we track the evolution of certain families of malware as they have grown and diversified, adapting and improving to effectively accomplish their required results.

Starting with the early variants we show how the malware has changed to meet the requirements of its producers, how the code is implemented to best fulfil those requirements, how the malware producers test their creations and finally how they deliver the final release to their involuntary users.

From this we draw conclusions regarding the various stages of the malware development life cycle and ascertain how their development reflects the strategies employed to produce the malware and what could be expected in future.

INTRODUCTION

The production of malware has become an industry that is healthy and growing at a steady pace. No longer is it just about an individual or two, or even a small group of like-minded miscreants, attempting to prove something. Today the primary motivation, as with any other industry, is commercial success.

To this end the producers of malicious software pursue their goal as rigorously as any producer of a large-scale software project. This results in a class of malicious software that is larger and more complex, is more resilient to detection and removal and modified and updated much more frequently [1].

The malware resulting from this process is often much harder to deal with and is more effective in accomplishing its goals.

Recent malware is designed not to be disruptive or intrusive but rather to stay under the radar and remain undetected while carrying out its nefarious activities for as long as it is required to do so. To achieve this purpose malware has evolved in size and complexity, split into multiple components and uses a variety of delivery mechanisms in order to infect a user's machine. The producers of malicious software have also employed a variety of protection measures which are under constant improvement and development, in an effort to remain undetected and still viable to the producers.

The production of such malicious software at a relatively high frequency requires a certain amount of development effort and in many cases a team of proficient producers to ensure the endeavour is successful.

FUNCTIONAL MALWARE DEVELOPMENT

In order for malware to be able to successfully infect a user's machine and carry out its intended purpose, work has been done to improve the chances of its success, which covers a wide range of techniques and procedures. This includes the ability to successfully deploy the malware onto the target machine, and once a beachhead has been established to then utilize various protection mechanisms to ensure its survival, in order to be able to deliver its payload. To ensure success each of these facets would require significant attention from the producers of the malicious software in question.

REQUIREMENTS TO ENSURE THE SUCCESS OF MALWARE

To ensure the success of malware in the current landscape the producers of said malware usually take certain steps when deploying their creations in the wild. First is to get the malware onto the target system, then steps have to be taken to ensure the survival of the malware on the target system, and finally once it has established itself the malware may engage its payload.

Delivery systems utilized by malware for initial machine compromise

There have been several often abused avenues of getting malware installed onto a target machine. Some of these include websites, compromised or otherwise, which may exploit vulnerabilities in software installed on a target machine. There are also actively spreading worms, which propagate via exploits, email, peer-to-peer networks and instant messaging, which may drop or download other malware as part of their payload.

In many cases this leads to some form of 'downloader' executing on a target machine. This usually means the installation of another or several different pieces of malware on the target machine resulting in a fan-out process of successive malware installation on a compromised target machine.

A growing trend has been not to target vulnerabilities in software but to exploit the users of that software. Social engineering plays a significant role in how malware gets installed. Malware producers trick and entice the user into downloading and executing their products for them. This leads to another issue in that success or failure of the endeavour depends entirely on the user, who is the vulnerability that is exploited, and in this case there is no patch that can be applied to fix the problem.

Often we see malware split into many functional components, and these components may arrive in the form of a package. There have been several different implementations utilized by malware to package one or more components, including binders, various types of dropper, and executable injectors.

Protection measures to ensure survival after a successful compromise

One of the major requirements of malware is to remain undetected and viable once installed on a target machine. There has been significant effort channelled into this area by malware producers, with varying degrees of success.

Some of the most often-used techniques include compression and encryption of malware binaries by freely available runtime executable packers. Depending on which ones are used, this method can give malware producers enough of a head start in the race.

Another popular and increasingly common technique is the use of code obfuscation to distort the flow of code, thus making it harder to analyse and implement detection for. Another technique gaining in popularity is the use of executable injectors, which inject executables into a given process; the injected executable is usually stored encrypted as part of the injector binary.

Many of these strategies also employ techniques such as anti-debugging and anti-emulation to further complicate the analysis and detection of malware that employs them. We have also seen the increased use of 'server-side polymorphism' [2], whereby many of the previously mentioned techniques are applied by themselves or in combination to the malware on the hosting server, and are then reapplied at regular intervals thus altering the appearance of the malware binary that is downloaded from a particular server at any given time.

Some other protection measures target the operating environment. Malware variants often check to ensure that they are not running in a virtual environment. To prevent removal, malware may often resort to several different strategies including but not limited to monitoring associated registry entries and files and replacing them if required, infecting or patching system files which when executed cause the malware to be dropped and executed and injecting code into system processes that prevent associated files from being removed.

Another measure that has enjoyed quite a bit of popularity is malware burrowing into the lower levels of the operating system. We have witnessed different techniques from user-mode hooking of APIs to the use of kernel-mode drivers, for the purposes of stealth, interception of information and bypassing system safeguards.

Payload: the goals of malware after a successful infection

Along with installation and protection the other major priority for the producers of malware is the reason they create their constructs in the first place, that is to deliver the payload. Depending on the requirement of the malware producers the payload can vary considerably. Some of the more popular items are as follows.

One of the most coveted items on a user's computer is the information it contains. This can include game data, banking information, personal data, account login information and logs that track user activity.

There appear to be a significant number of families whose primary payload is to use the compromised machine to send out bulk unsolicited email. In many cases the compromised machine can be controlled remotely by an attacker through the use of commands sent to the installed malware, thus enabling the attacker to use the compromised machine to do their bidding.

Another target of malware installed on a compromised machine is none other than the users themselves. There is a growing crop of malware that uses social-engineering techniques to intimidate, scare, or annoy the user into purchasing dubious software products.

IMPLEMENTATION AND IMPROVEMENTS TO REQUIREMENTS AS SEEN IN THE WILD

Malware implementations vary according to the needs of producers but there are also several common methods and techniques utilized by malware producers to implement their solutions. This is because often there are a limited number of solutions to a problem and there is also a substantial amount of sharing and reuse of code that takes place. We take a look at certain implementations of some of the previously mentioned requirements.

Example implementations of delivery systems

Most delivery systems today rely on malicious software whose main purpose is to download and execute other arbitrary pieces of code. This type of software is prolific and often one of the first pieces of malware to be run on host machines. We have seen many and varied types of implementations of such malware, written in a plethora of different programming languages. Here we take a look at a few different types of implementations.

Because of the limited scope of the problem that is addressed by this type of malware, that is, to download and execute code from a remote site, it is often implemented as a small binary. This is especially true if it is implemented in Assembler. One example is a family of downloading trojans we call 'Small.gen!O'. This trojan is usually under 4 KB in size and its function is quite straightforward, with calls to standard APIs to download a file to the temporary directory and execute it. An attempt is made to hide its motives by obscuring the URL from which the file or files are downloaded by storing it at the end of the malware file using some simple encryption.

Once an implementation exists it often fulfils the requirements of the producers quite well, and the only room for improvement comes back to how to successfully it evades detection and removal from a compromised system before the job is done. Thus efforts have been made to improve in these areas. One such technique we have seen involves using the 'Background Intelligent Transfer Service' [3] to download files rather than standard APIs. One such trojan is 'Jowspry'; it uses the service by creating an instance of the BackgroundCopyManager COM object and creates a job of type 'BG_JOB_TYPE_DOWNLOAD', adding the required details to download the file. This technique may be used to disguise traffic as legitimate to bypass a firewall.

One downloader that has been quite prolific in the past few years is the Vxidl family. Vxidl is used to download several other pieces of malware onto a targeted system. Early variants of Vxidl started out with a straightforward implementation that downloaded files from URLs stored in the binary by calling standard APIs, imported dynamically before use, in a thread. Later variants saw functionality added that obtained locale information on the infected machine which was used to download a different binary depending on the locale. Some of the country checks included Australia, Canada, France, Germany, Spain, the United Kingdom and the United States. A further enhancement included obtaining machine-specific information such as processor type and OS version and posting this data to a remote site. This collected data then began to include the locale information, presumably to move the country selection functionality to a remote server.

More recent variants also include code to disable the *Windows Task Manager* and to add themselves to the *Windows Firewall* list of allowed programs. They also check for the presence of the *Outpost* firewall and report this information back. Some variants of Vxidl dropped a driver that could disable many AV products and was also used to hide the trojan's associated file and registry changes. One point of interest is that we first saw this driver used with variants of the Nuwar family, and later with other families such as Clodpuntor and Renos. At one point variants of Vxidl also included functionality to inject code into the svchost.exe process in order to download files. From very early on in their development Vxidl binaries were distributed packed, usually with FSG, but later variants use what has come to be known as 'Tibs' encryption.

Delivery mechanisms are used as a means to an end. There are many different methods by which the end result can be achieved, be it as a web page with an embedded IFrame that links to some malware, or an animated cursor exploit that downloads malware. It could be a fake video codec or an application that provides free access to pornography. Or even executables bundled with music files distributed on peer-to-peer networks. Any method that meets the straightforward requirement of getting malware onto a machine will be utilized, and when something that works well is found, such as social engineering, it is used often and repeatedly.

Examples of protection measures

One of the most actively pursued endeavours is how to protect a malicious program from detection and removal. One of the overriding goals of the producers of malware is to ensure the success of their product, which means using any and all measures available to protect their applications. Various types of protection methods have been utilized by malware over the years, and we have witnessed substantial development in this area.

Packers and cryptors

The use of executable encryption and compression software in malware has enjoyed a long and popular tradition. In this area the producers of malware are spoiled for choice and use is varied, from the latest packer 'du jour' to old favourites. Often these may be used in combination with each other. There has also been a rise in numbers of custom packers and cryptors used specifically by malware and distributed via underground channels. Some of these even blur the line between protection software and malware themselves, for example by giving the user the option not just of encryption but the option to bypass a firewall or other security software.

Server-side polymorphism

Another issue that is associated with exe packers and cryptors is how they are used. Traditionally new variants are released either repacked or re-encrypted with the same packer or something new, but this combined with a fast release cycle means that essentially the same binary underneath can be used as long as the outer shell cannot be breached. Taking this method to an extreme is the use of server-side polymorphism, whereby binaries are constantly changed, and may be different with each instance that is downloaded, although the frequency of changes may differ from one implementation to another. These changes may be due to the use of one or more different packers, or re-encryption with a custom cryptor that undergoes frequent changes such as the 'Tibs' cryptor that is used by variants of the Vxidl, Nuwar and Renos families amongst others. There are also variants of the Zlob family that have their binaries altered and repackaged on a daily basis.

Code obfuscation

Another protection measure is the use of code obfuscation. Many different types of obfuscation have been used in the wild. One type of code obfuscation utilizes the insertion of junk or meaningless instructions in between real instructions. This type can be seen in families such as Fanop and Zuten [4]. Variants of the Fanop family included instructions that when executed, effectively did nothing; these included floating point instructions.

This same method was then taken a bit further by variants of the Zlob family. Variants of Zlob inserted calls to APIs with incorrect parameters but would still allow normal execution to continue. Another method used by a family of trojans known as Stresid was to reorder code to hide the execution flow. The code was separated into short blocks, these blocks were then shuffled or rearranged and connected with jumps to maintain code flow.

A recent example of code obfuscation can be seen in variants of the family known as Pugeju [5], where the code is interspersed with SSE2 instructions and code flow is maintained by calling a redirect/jump function which alters the execution path to a new block of code. These types of technique are gaining in popularity and there are many other examples of malware that use similar methods, with their increase in use matched by an increase in complexity.

Anti-debugging and anti-emulation

Several protection measures target the mechanisms used by vendors to detect malware. These methods include anti-debugging and anti-emulation tricks. Some of these include calling the IsDebuggerPresent() API to check if a debugger is running; however, there are also several other methods for detecting if a debugger is running, several of which are detailed

in [6]. In many cases the program that is debugged might simply quit, or sometimes the malware might decide to take an alternative course. One such example can be seen in variants of the DelfInject [7] family, which use instruction timing. If a debugger is detected the code can jump to an invalid address or end in an infinite loop.

Variants of the Gnorug family used a cryptor which employed a strategy of checking for the debugger using the NtQueryInformationProcess API as detailed in [6]. The returned result is XORed with an existing decryption key. If a debugger is detected the result will be the value 0xffffffff, which then causes the rest of the code to decrypt incorrectly.

There are also several anti-emulation techniques that have been used to thwart the use of emulation in detection by vendors. For instance, the use of delay loops which achieve nothing other than to increase the amount of time it takes for the execution of a certain bit of code, say a decryption routine, to complete. Another technique which has gained popularity is the use of operating system specific properties to alter how a section of code executes, such as using the return values of APIs called with particular parameters as a condition to determine how execution should proceed or as a key required in decryption. This has been taken a step further by malware that relies on the implementation side effects of certain APIs in a specific operating system version to proceed with further execution. Another popular method is the calling of obscure or unrelated APIs which do not have any bearing on the code that is currently executing and may be handled incorrectly by an emulator.

System monitoring

In order to ensure appropriate protection measures malware has to take into account the execution environment. There have been several implementations that target the operating environment in order to protect themselves. One such method is the constant monitoring of registry entries and files that are associated with the malware. These are then replaced if removed. This method has been extended by usually injecting the code that performs the monitoring into a system process, thus making it harder to remove. There have also been variants of malware that infect/patch system files that enable malicious files to be dropped and executed when the patched system code is executed. Depending on the infection strategies used, this can be every time a patched system function from the infected binary is called.

Classic stealth

Another group of tactics that are quite often utilized are the classic stealth techniques [8]. Many have been used in the wild, and can range from something as simple as turning on the hidden and system file attributes, to hooking relevant APIs with code that hides the associated file, registry and process information. User-mode hooking has also been supplemented with kernel-mode hooking via the use of device drivers [9]. Initially drivers were used primarily for stealth, but then kernel-mode code was also used for other forms of protection such as termination and removal of AV and other security programs, as well as bypassing firewalls.

Anti-analysis

Finally, malware producers are often concerned with where their creations are executed. For example, they may not want their creations to execute in a lab environment or honeypot system. There have been efforts made to discover if malware is executing in such environments. This may include checks to see if something is executing on a virtual machine. There have also been several methods used to accomplish this, including using the SIDT instruction and similar, where checks are made that are specific to certain virtual environments such as *Virtual PC* and *VMware.* Several are described in [10].

Examples of payload implementations

The 'payload' is usually the end goal for malware, the reason it was created in the first place, that brings its creators the most benefit. Popular and profitable payloads today include the sending of spam, and the stealing of sensitive and valuable information.

Sending spam

There is a large proportion of malware whose primary purpose is to use the infected machine to send out spam. Although the techniques utilized to accomplish this may have changed, the basic approach remains the same.

The usual method involves downloading a template of the messages that are to be sent along with all the other relevant details that are required to send out a batch of emails. Once a run has been completed the malware may send back statistics on the previous run. Variants of the Chopanez family utilize this approach and communicate with the attacker via http; there are of course several variations on this theme. More recently, trojans that send spam have become more complex. An example of this is the Nuwar [11] family which has the ability to receive information used in a spamming run via a peer-to-peer network.

Several of the spamming trojans use kernel-mode code. The main use is as part of a stealth component, in the form of a driver. Examples include variants of Fanop and variants of Cutwail [12]. The Srizbi [2] trojan takes this to yet another level by implementing nearly all of its functionality using a kernel-mode driver. In addition to sending spam, many of these trojans can also be used as a platform for malware delivery.

Stealing sensitive data

The importance of information stored on the average personal computer may often be overlooked, but there is a significant amount that can be used as currency by criminals. Information that is targeted by thieves may include software activation keys, authentication information, game-related information, as well as personal and financial data. Malware has used several methods to obtain such information. These run the gamut from targeting specific online games for information as can be seen in trojans such as Zuten, to others that hook network-related APIs sniffing for information as can be seen in trojans such as Ursnif. An interesting approach was used by early variants of the Sinowal [13] trojan which contained an implementation for regular expression search. This was used to search browser memory for banking information. Even though Sinowal's implementation

has changed and the protection measures it uses are vastly different today, this trojan still targets banking information.

Malware is created to deliver a payload, thus it appears that once an implementation of a particular payload is refined, it usually remains the same. There may always be minor improvements and refinements to be made, such as when a new version of a game or a different application is targeted, but for the most part once the malware producers have found a method that works they tend to stick with it.

TESTING OF MALWARE IMPLEMENTATIONS

Like most software producers, malware authors test their creations for bugs. Although this may not be as stringent a requirement as it is with other software producers, it is required in order to ensure that the resulting software performs at an adequate level of proficiency.

The finding and fixing of bugs may be a necessary requirement but appears not to be the primary concern of malware producers when testing their creations. The primary goal of testing is to ensure that the malware that is produced is able to evade detection by any anti-virus scanner which is also one of the reasons for constant updating of malware as mentioned in section 7.2 of [14] or security software that is installed on the machines they wish to target.

It is very difficult to ascertain exactly how malware authors test the viability of their creations to evade detection mechanisms used by anti-virus software, but we can draw conclusions by observing certain binaries which appear to have originated from particular groups that produce malware.

One method that may be employed is to obtain copies of widely used anti-virus scanners and security software and install them in a test environment which enables them to check detection rates. However this is not always a feasible method of testing. Another is to utilize one of the many online scanning services that are available to provide feedback on submitted executables.

If scanning does detect the malware binary then steps must be taken to find a solution. This usually means altering the binary in some way. In previous years this may have involved packing or encrypting the binary with one or more executable packers or cryptors (as discussed previously). The effectiveness of these can also be tested by using them on malware that is already widely detected and checking the results.

Recently, however, we have observed more sophisticated testing methods used by malware authors. These methods target the detection technologies utilized by the anti-virus scanners and other security software, with the use of carefully crafted binaries to determine where and how a file is detected and in some instances what detection method was used. We have observed altered binaries from the producers of families such as Nuwar, Cutwail and Pugeju.

Testing for detection of their products does appear to play a significant role in the production of malware. There is also nothing stopping malware producers from reverse engineering anti-virus scanners to gain knowledge of detection methods to improve their testing practices.

CONCLUSION

Malware has increased in complexity and more sophisticated techniques are gaining widespread use, increasing the effort required for combating this problem. The producers of malware will continue to use whatever means necessary in order to deliver their creations to their intended targets. This means looking for new avenues while still using every other available method.

The techniques used to protect malware have also seen a significant increase in complexity and we have seen a parallel increase in the use of custom solutions as opposed to 'standard off-the-shelf' measures. This indicates more investment in development by malware producers. In the end, the goals of producers have remained consistent and the methods used to perform payloads adhere to well tried and tested techniques.

If the trend continues, we can expect to see a lot more widespread use of many of the more sophisticated techniques used to protect malware, including an increase in experimentation with the use of various new techniques and protocols in deployment, protection and command and control of malware, combined with an accelerated release cycle, and all with the goal of sending more spam, stealing sensitive information and misleading users for gain.

REFERENCES

[1] Molenkamp, S.; O'Dea, H. Solving the Bagle jigsaw. Proceedings of the Virus Bulletin International Conference, 2005.

[2] Molenkamp, S.; O'Dea, H. Have you got anything without spam in it? Proceedings of the Virus Bulletin International Conference, 2007.

[3] About BITS. http://msdn.microsoft.com/en-us/library/aa362708.aspx.

[4] PWS:Win32/Zuten.gen!A. http://www.microsoft.com/security/portal/Entry.aspx?name=PWS%3aWin32%2fZuten.gen!A.

[5] Trojan:Win32/Pugeju.gen!A. http://www.microsoft.com/security/portal/Entry.aspx?Name=Trojan%3aWin32%2fPugeju.gen!A.

[6] Ferrie, P. Anti-unpacking tricks. CARO Workshop, May 2008, Amsterdam. http://pferrie.tripod.com/papers/unpackers.pdf.

[7] Trojan:Win32/DelfInject.gen!D. http://www.microsoft.com/security/portal/Entry.aspx?Name=Trojan%3aWin32%2fDelfInject.gen!D.

[8] Kasslin, K.; Ståhlberg, M.; Larvala, S.; Tikkanen, A. Hide 'n seek revisited – full stealth is back. Proceedings of the Virus Bulletin International Conference, 2005.

[9] Hoglund, G.; Butler, J. Rootkits Subverting the Windows Kernel. Addison-Wesley, 2006.

[10] Ferrie, P. Attacks on virtual machine emulators. Proceedings of the AVAR Conference, 2006.

[11] Backdoor:Win32/Nuwar.C. http://www.microsoft.com/security/portal/Entry.aspx?Name=Backdoor%3aWin32%2fNuwar.C.

[12] Spammer:Win32/Cutwail.gen!B. http://www.microsoft.com/security/portal/Entry.aspx?Name=Spammer%3aWin32%2fCutwail.gen!B.

[13] PWS:Win32/Sinowal.gen!D. http://www.microsoft.com/security/portal/Entry.aspx?Name=PWS%3aWin32%2fSinowal.gen!D.

[14] Ször, P. The Art of Computer Virus Research and Defense. Symantec Press, 2005.

Appendix

Sample instructions from Fanop:

```
7A 00           jp      short $+2
EB 00           jmp     short $+2

60                      pusha
61                      popa

F7 D2                   not     edx
F7 D2                   not     edx

DD C5                   ffree   st(5)
DD C1                   ffree   st(1)

D9 D0                   fnop

9B DB E2                fclex
```

Sample instructions from Zuten:

```
90              nop
8B C9           mov     ecx, ecx
90              nop
8B C0           mov     eax, eax
90              nop
8B C9           mov     ecx, ecx
90              nop
90              nop
8B C9           mov     ecx, ecx
8B D2           mov     edx, edx
90              nop
8B D2           mov     edx, edx
```

Sample instructions from Zlob:

```
.text:07541961 6A 00        push    0     ; hThread
.text:07541963 FF 15 54 30 54 07   call  ds:
ResumeThread
.text:07541969 6A 00        push    0     ; lpOverlapped
.text:0754196B 6A 00        push    0     ;
lpNumberOfBytesRead
.text:0754196D 6A 00        push    0     ;
nNumberOfBytesToRead
.text:0754196F 6A 00        push    0     ; lpBuffer
.text:07541971 6A 00        push    0     ; hFile
.text:07541973 FF 15 58 30 54 07   call  ds:ReadFile
.text:004010A5 FF D6        call    esi   ; PulseEvent
.text:004010A7 6A 00        push    0     ; hLibModule
.text:004010A9 FF D7        call    edi   ; FreeLibrary
.text:004010AB 6A 00        push    0     ; lpProcName
.text:004010AD 6A 00        push    0     ; hModule
.text:004010AF FF D3        call    ebx ;
GetProcAddress
```

Sample instructions from Stresid:

Block 1:

```
.text:004013ED E8 8B 01 00 00     call sub_40157D
(Block2)
.text:004013F2 3D 00 00 00 D0     cmp eax, 0D0000000h
.text:004013F7 0F 87 71 FF FF FF  ja loc_40136E
.text:004013FD FF 25 47 20 40 00  jmp off_402047 (Block
4)
```

Block 2:

```
.text:0040157D 29 C0              sub eax, eax
.text:0040157F 50                 push eax
.text:00401580 0F 01 4C 24 FE     sidt fword ptr
[esp+4+var_6]
.text:00401585 58                 pop eax
.text:00401586 FF 25 7C 21 40 00  jmp off_40217C
(Block 3)
```

Block 3:

```
.data:0040217C 41 12 40 00        off_40217C dd offset
sub_401241
.text:00401241 C3                 retn
```

Block 4:

```
.data:00402047 34 10 40 00        off_402047 dd offset
sub_401034
.text:00401034 3D 00 00 00 80     cmp eax, 80000000h
.text:00401039 0F 83 09 07 00 00  jnb loc_401748
.text:0040103F FF 25 AC 21 40 00  jmp off_4021AC
```

Sample instructions from Pugeju:

```
.text:00405B04 C7 05 21 F2 40 00 01 00 00 00       mov
dword_40F221, 1
.text:00405B0E 0F F8 E2           psubb mm4, mm2
.text:00405B11 C7 C7 05 00 F1 40 00 30 75 00 00    mov
dword_40F100, ,u0‘
.text:00405B1B E8 D3 E3 FF FF     call sub_403EF3 (call
to jump function)
.text:00405B20 C2 08 00           retn 8
.text:00405B23 FF         db 0FFh
.text:00405B24 00         db 0
.text:00405B25 00         db 0
.text:00405B26 00         db 0
.text:00405B27 00         db 0

.text:00405B28 E8 A3 00 00 00      call sub_405BD0 ←
(jump function returns here)
.text:00405B2D 83 3D FB F7 40 00 00       cmp dword_
40F7FB, 0
.text:00405B34 74 4F              jz short loc_405B85
.text:00405B36 0F DC E1           paddusb mm4, mm1
.text:00405B39 6A 02              push 2
.text:00405B3B 6A 00              push 0
.text:00405B3D 6A 00              push 0
.text:00405B3F 0F 6B C8           packssdw mm1, mm0
```

```
.text:00405B42 E8 AC E3 FF FF     call sub_403EF3
.text:00405B47 C2 08 00           retn 8

jump function:
.text:00403EF3 87 04 24           xchg eax, [esp]
.text:00403EF6 8D 40 04           lea eax, [eax+4]
.text:00403EF9 0F F5 DD           pmaddwd mm3, mm5
.text:00403EFC 03 00              add eax, [eax]
.text:00403EFE 8D 40 04           lea eax, [eax+4]
.text:00403F01 87 04 24           xchg eax, [esp]
.text:00403F04 0F EC F0           paddsb mm6, mm0
.text:00403F07 C3                 retn
```

Sample instructions from Delfinject:

```
CODE:00402AED 0F 31               rdtsc
CODE:00402AEF 8B C8               mov      ecx, eax
CODE:00402AF1 0F 31               rdtsc
CODE:00402AF3 2B C8               sub      ecx, eax
CODE:00402AF5 F7 D1               not      ecx
CODE:00402AF7 81 F9 00 00 01 00   cmp      ecx, 10000h
CODE:00402AFD 7C 05               jl       short loc_
402B04
CODE:00402AFF E9 15 84 C8 F8      jmp      near ptr
0F908AF19h ← (invalid jump)
CODE:00402B04
CODE:00402B04             loc_402B04:
CODE:00402B04 0F 31               rdtsc
CODE:00402B06 8B C8               mov      ecx, eax
CODE:00402B08 0F 31               rdtsc
CODE:00402B0A 2B C8               sub      ecx, eax
CODE:00402B0C F7 D1               not      ecx
CODE:00402B0E 81 F9 00 00 01 00   cmp      ecx, 10000h
CODE:00402B14
CODE:00402B14    loc_402B14:      ; CODE XREF: sub_
4029E8:loc_402B14j
CODE:00402B14 7F FE               jg       short loc_
402B14  ← (Inifinte loop)
CODE:00402B16 EB 02               jmp      short loc_
402B1A
```

Sample instructions from Gnorug:

```
.text:0041272B 31 45 3C           xor [ebp+3Ch], eax <-
(Xor Result with key)
.text:0041272E E8 00 00 00 00     call     $+5
.text:00412733 5E                 pop      esi
.text:00412734 83 C6 12           add      esi, 12h
.text:00412737 8B FE              mov      edi, esi
.text:00412739 B9 6C 00 00 00     mov      ecx, 6Ch
.text:0041273E
.text:0041273E                    loc_41273E:
.text:0041273E AD                 lodsd
.text:0041273F 33 45 3C           xor      eax,
[ebp+3Ch] <- (Use key to decrypt)
.text:00412742 AB                 stosd
.text:00412743 E2 F9              loop     loc_41273E
```