

Malicious Malware: attacking the attackers, part 2

Thorsten Holz, Frederic Raynal 2006-02-02

This article explores measures to attack those malicious attackers who seek to harm our legitimate systems. The proactive use of exploits and bot networks that fight other bot networks, along with social engineering and attacker techniques are all discussed in an ethical manner. Part two of two.

Introduction

In [part one](#) we discussed the first two of four main objectives one has in fighting back against malicious hackers. We started with information gathering - by deceiving a malicious attacker and subverting his attacks. Then we looked at preventing access to some resource on the attacker's system (Denial-of-Service) through the use of resource starvation and fake exploits that the malicious individual has not fully reviewed.

Now in part two we finish the discussion by looking at how to own a malicious attacker's precious resources, which he planned to use against us, and then use these resources as a stepping stone to fully achieve our goals. If the reader has not fully read [part one](#) of this article, he is encouraged to do so now.

Owning the attackers

Our next goal is to take control of a malicious attacker. Of course, we could have used the previous backdoored exploits to do so. It has been reported that some exploits could be exploitable, so installing a fake vulnerable target which hacks back the exploit is certainly possible. However, this "sport" is especially interesting among malware.

Malware exploiting malware: parasitic propagation

Today we are also faced with malware that takes advantage of anterior malware. This could be either the use of backdoors left by an exploit or even the exploiting of vulnerabilities on another piece of malware. In this section, we will give several examples of this kind of species and show how this can be abused to exploit an attacker.

Presumably the most prominent example of this category is DoomJuice. This piece of malware tries to propagate further with the help of the backdoor that was left by MyDoom, one of the worms spread quite widely. The worm MyDoom was first observed at the end of January 2004. It primarily propagates further with the help of e-mails, sending itself to e-mail addresses found on a compromised machine. In addition, it also uses the Peer-2-Peer application KaZaA to spread even further. The interesting payload of this malware is the backdoor it opens on TCP port 3127. The backdoor is created on each infected host, causing them to be an easy target for further attacks. The second payload of MyDoom was a Distributed Denial-of-Service (DDoS) attack against SCO and Microsoft, one that caused only a limited amount of damage. MyDoom was quite effective worm at the time, from an attacker's point of view. For example, MessageLabs reported that at the peak of its spreading, 1 of ever 12 e-mails was infected with the worm, resulting in MyDoom being the fastest worm spreading ever via e-mail. In the first 24 hours, MessageLabs has intercepted more than 1.2 million copies of this worm. [[ref 5](#)]

Then at the beginning of February 2005, the first malicious malware tried to take advantage of the

huge number of MyDoom-infected systems. The worm DoomJuice is a parasitic worm that uses the backdoor opened by MyDoom to propagate further. So this worm actually exploits a backdoor that was left by an anterior worm, resulting in an effective propagation mechanism without much additional effort. DoomJuice itself was not that interesting of a worm, since the payload is rather boring: it just tried to cause a Denial-of-Service attack against Microsoft. Presumably much more harmful consequences could have been possible with such a worm. This worm is just one example of a whole class of parasitic malware that exploits other kinds of malware - often resulting in quite effective propagation from an attacker's point of view.

As another example, let's take a look at the Sasser worm. This worm started to spread across the Internet at the end of April, 2004. It exploited a vulnerability announced in Microsoft Security Bulletin MS04-011, the well-known LSASS vulnerability that is a stack-based buffer overflow in some functions of LSASRV.DLL. Since it was one of the first worms that used this vulnerability, it was able to compromise a very large number of different systems.

However, since programming without errors is quite hard, there were also vulnerabilities found in Sasser. In this particular example, there was an error in the implementation of the FTP server. This server is mainly used to transfer the binary itself from one machine to another, thus helping it propagate. The FTP server of Sasser listens on TCP port 5554 and contains a stack-based buffer overflow - after all, it is just software written by a human. The interesting thing about this is that now there is also parasitic malware that takes advantage of the huge Sasser population in the wild. There are several exploits available to exploit this vulnerability and in addition, there is also a worm that targets hosts infected by Sasser. This worm is called Dabber [ref 6] and it also uses a parasitic propagation mechanism. This exploitation of an actual worm is an interesting example of a spreading mechanism. Similar to DoomJuice, Dabber just takes advantage of the huge population of a certain malware and uses this to spread further. Dabber itself sets up an authentication backdoor on each compromised system that can be used by the attacker to take control over the system.

"This is not a viruswar, this is a botwar!"

In the area of bots, there are also several examples of malware that fight against other kinds of malware. If a computer is not on the latest patchlevel, it can often be exploited. This is especially true for computers running Windows as an operating system without essential patches like Service Pack 2 for Windows XP. These systems are an easy target for autonomous spreading bots: our experience, with the help of honeypots, shows that it takes on average only a couple of minutes for a system that is vulnerable to MS03-026 (DCOM) or MS04-011 (LSASS) to be successfully infected with a bot. Most bots implement a command to "secure" the exploited machine. This is mainly done by disabling network shares or vulnerable services. Botnet controllers often use this command to prevent a machine from being infected with more than one bot. But if the controller does not issue this command, the machine can be infected with several bots and then the fight begins.

An interesting example is given by F-Secure. [ref 7] They have a nice little drawing of several families of bots that try to stop each other. All of the bots exploit the recent Plug and Play vulnerability (MS05-039) and were among the first to do so. In order to create a large botnet, the controller of each different version of the bots wants to have complete - and exclusive - control over the infected machine. The easiest way to achieve this is to look for signatures of other bots. If an instance of another bot is found, the bot tries to stop the process of the other bot or even deletes the competing bot.

Offensive deception

Deception can also be used as part of an offensive tactic. We can preserve our resources and yet waste some of the resources of our opponent. This can be achieved by building up a solution of deception,

one that causes the attacker to spend a larger amount of his resources to explore our camouflage. This is what happened during the Second World War: many fake troops were disposed at several points on the English coasts, such that the German army could not guess the exact location of attack on D-day.

The Allies have used deception to gain an advantage during their offensive. This worked because the German army (defense) knew something was going to happen and they were expecting it. This is a necessary condition to use deception to attack: the target needs to expect to be attacked. Of course, creating this expectation can be part of the tactic itself... this is particularly true here. And usually, the ones expecting to be attacked are the system and network admins, not the attackers themselves.

A very good example is given by Laurent Oudot [ref 8] where an honeypot is used to detect hosts infected by a worm. When the worm tries to spread to the honeypot, it means the source is infected, and thus needs to be patched because the flaw is present. Hence, the flaw is exploited by the honeypot, which then patches the infected host.

For a few years now, we have also had another playground for offensive deception: wireless networks. These are (almost) an open network, even when using WEP. So, the first step we can take to deceive the attacker is to use a program that [creates many fake APs](#) (an example is given with scapy [ref 9], below):

```
>>>sendp(Dot11(addr1="ff:ff:ff:ff:ff:ff",addr2=RandMAC(),addr3=RandMAC())/
Dot11Beacon(cap="ESS")/Dot11Elt(ID="SSID",info=RandString(RandNum(1,50)))/
Dot11Elt(ID="Rates",info='\x82\x84\x0b\x16')/
Dot11Elt(ID="DSset",info="\x03")/
Dot11Elt(ID="TIM",info="\x00\x01\x00\x00"),
iface="wlan0ap",loop=1)
```

Someone coming around and trying to detect our wireless network will not discover a single AP but instead, many - too many. We can also broadcast other frames than just beacon ones. For instance, we can send frames containing exploits against typical tools used on such networks: kismet has had a few flaws fixed lately, and many flaws were found in ethereal. So imagine what happens if you let your wireless network send beacons and exploits to the outside world. When someone attempts to intrude into your network, he will succeed... but so will our attack too: we will just have to wait for the target to reconnect from somewhere else, and our exploit will "phone home." Some proof of concept code for this is available in KARMA. [ref 10] The purpose is simply to provide a fake environment for wireless clients, and then exploit client side flaws.

These two examples are showing how to use deception against an unidentified attacker, in a reactive way when they attempt to perform their offensive against us. But at least this approach targets attackers as they are attempting to intrude into our systems. Of course, this method can be improved by adding some information gathering before attacking back, for instance by using passive scanning to guess the attacker's system or similar techniques.

Stepping stones

Now in our final section, let's see how we can use an attacker as a stepping stone. This supposes that he has already performed his job and has attempted to attack us.

Attacking an attacker to own his targets

Our first example is based on the rootkit called SucKIT [ref 11], and recent sources are available. [ref 12] We present a *hack back* technique to target an attacker that has attacked one of our machines. We want to reactively stop somebody who has attacked us with SucKIT. This rootkit uses authentication and cipherring to preserve its configuration. Here, we will try to retrieve SucKIT's configuration so that we can use it to get to other hosts already compromised by the intruder: a single SucKIT binary can be a really good way to step into a network of compromised hosts. This example is built upon a real-life example. [ref 13]

Imagine a situation where an unknown binary is found (called `binary` later) on a compromised host. Before analyzing the information, let's see how SucKIT works. The configuration is saved in a structure called `config` (include/config.h):

```
// Taken from include/config.h

struct config {
    char    home[256];
    char    hidestr[16];
    uchar   hashpass[20];
} __attribute__((packed));
```

This structure is ciphered in the binary using RC4. During SucKIT's compilation, a random seed is used to initialize the stream cipher so that each installation is different. When SucKIT is run for the first time, the RC4 seed and the above structure are saved at the end of the binary. The structure is ciphered with RC4 and is 356 bytes long, the 64 first bytes being the RC4 seed.

However, when it is run, SucKIT starts by deciphering the binary loaded into the memory. Hence, we just have to dump the memory of the process to get our information as clear text (we will not explain here how to dump memory - finding the location of the structure using binary differentiation is out of the scope of this article):

```
$ gdb -q -p `pidof binary`
Attaching to process 575
/proc/575/exe: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) x /s 0x5debcaba
0x5debcaba:      "/usr/share/locale/.dk20"
(gdb) x /s 0x5decbba
0x5decbba:       "dk20"
```

So, now we just need the last (but not least) piece of information: the hash of the password. It is located right after this in the memory of the process:

```
(gdb) x /20bx 0x5decbca
0x5decbca:      0x77    0xa0    0x56    0x93    0x5a    0xba    0xb3    0x29
0x5decbd2:      0xf4    0xf3    0x18    0x2f    0x42    0xee    0xd8    0x86
0x5decbda:      0x76    0xc7    0x24    0x47
```

Now that we have the hash of the password, we could try to break it. However, this could take a very long time or even be impossible, as it depends mainly on the hash function and charset used in the password (in fact, in our example it took one full day to retrieve the real password).

But there is a much more clever option to own a 'SuckITed' network of hosts. We can patch the source of the rootkit so that no authentication other than the hash is required. For instance, with a client version 1.3, the patch in `zlogin.c` is:

```
+char hashpass[] = "\x77\xa0\x56\x93\x5a\xba\xb3\x29\xf4\xf3\x18\x2f\x42"
+                "\xee\xd8\x86\x76\xc7\x24\x47"

- hash160(p, strlen(p), &h);
+ /* hash160(p, strlen(p), &h); */
+ memcpy(h.val, hashpass, sizeof(h.val));
```

Once this is done, we can rely on the SuckKIT client to connect to owned systems. The problem is now to identify them. If the intruder is not meticulous, he may have connected to the current host from another compromised host. This is a first target on which we can try our patched client (right after localhost). But if our intruder is not meticulous, and moreover is inherently lazy, he will use the compromised host to connect to other targets. Remember that SuckKIT embeds a sniffer... and the file `.sniffer` will contain all the information we need. We know where this file is located on the system because we have the configuration of the rootkit. Thus, we will see all the connections made to remote hosts.

Attacking an attacker to target another attacker

Another example in this area is the hostile take-over of *botnets*. As we know, a botnet is a network of compromised machines that can be remotely controlled by an attacker. Nowadays, botnets are a huge threat: with the help of a large, remote controlled network, attackers can control several hundred or even thousand bots in parallel, thus enhancing the effectiveness of their attack. Botnets in particular pose a severe threat to the Internet community, since they enable an attacker to control a large number of machines. Attackers primarily use them for attacks against other systems, mass identity theft, or sending spam.

The fun part starts when we want to take over the very botnet that an attacker controls. This is mainly done in the following way: first, we capture one of the bots of the attacker. This is rather simple since the bots are constantly spreading, and it is easy to get a sample of a bot. Via reverse engineering, we can retrieve all of the sensitive information of a botnet. This information includes the DNS name or IP address of the central server that is used for Command & Control (C&C server) and its port number. Furthermore, we need the IRC name and channel used by the bots, the ident structure, and (if necessary) passwords used to connect to the C&C server and to enter the channel.

With this information, we can now impersonate a valid bot and become part of the botnet. This allows us to observe everything inside the botnet: we can see the commands issued by the attacker, estimate the size of the botnet if the C&C server is not configured to prevent this, and more. Eventually, we will also observe the password the attacker uses to authenticate himself to the bots. We can also retrieve this password via reverse engineering, but just observing him entering his password is easier and more fun. If this password is the only authentication mechanism used in the botnet, we can use it to take control over the bots. We could, for example, install our own bot on the machines, thus taking control over them.

As an example now, we want to present an observation of a real botnet. Via reverse engineering, we found out all necessary information to connect to the botnet and impersonate ourselves as a valid bot, as explained above. This allows us to also observe the attacker login in and starting issuing commands to the bots to propagate further:

```

14:53 < f0rt3> .login c1cer0
14:53 < Hack-0311> A vos ordre mon maître !!!
14:53 < Hack-9908> A vos ordre mon maître !!!
14:53 < Hack-5620> A vos ordre mon maître !!!
14:53 < Hack-0737> A vos ordre mon maître !!!
14:53 < Hack-5835> A vos ordre mon maître !!!
[...]
14:53 < Hack-1201> A vos ordre mon maître !!!
14:53 < Hack-8152> A vos ordre mon maître !!!
14:53 < f0rt3> .ntscan 1000 10000 200 -b
14:53 < Hack-0341> [NTScan] Scan maintenant 1000 threads pendants 10000 minutes de 200
14:53 < Hack-0931> [NTScan] Already scanning
14:53 < Hack-1918> [NTScan] Scan maintenant 1000 threads pendants 10000 minutes de 200
14:53 < Hack-5720> [NTScan] Scan maintenant 1000 threads pendants 10000 minutes de 200

```

We could now try to use the password "c1cer0" to authenticate ourselves to the bots. If no other protection mechanism is in place, this will be successful and we could then command the bots however we like. This is a rather pro-active way to stop a dangerous botnet, and inhibit the attacker from causing more harm with it.

Conclusion

We have presented several examples of malicious malware, by showing exploits that attack the ones who execute the exploits against us, or how bots can fight against other bots. In addition, we have illustrated how attackers have targeted other attackers by exploiting programming flaws in other malware (known as "parasitic propagation") and how fake exploits are sometimes used in clever ways to exploit the lazyness and curiosity of humans. This article should raise awareness of the dangers of executing any code that has not been fully analyzed and understood. In addition, the authors hope to have come across with a positive, entertaining article that may even have caused the reader to enjoy our approach.

How does one defend himself and his network against this kind of malicious malware? First and foremost, one has to analyze the code he is going to execute. Take a close look at *what* is executed *when*, and even take a look at possible vulnerabilities in the given source code. Secondly, analyze the shellcode and understand what it does. If it contains the sequence `rm -rf`, for example, you should be very suspicious since normally an exploit does not contain command to wipe a hard disc. There are also tools like fakebust [ref 14] that check a given shellcode or binary for suspicious sequences of commands. These kind of tools can help to mitigate risk. Examine the given shellcode in the exploit with these tools, do a manual analysis, and you should certainly be more secure. Moreover, it is recommended that one use some kind of virtual machine to play with any unknown code you examine. This way, you can at least mitigate some of the risk. Of course, the program may behave differently if it is executed within a virtual machine, so take care when dealing with malware!

The world of malware, or malicious software, is becoming a real eco-system unto its own, with predators feeding off each other. Some "creatures" eat others, abuse them or just act as parasites to them. We are approaching the next level. Perhaps the Matrix is not that far off after all.

Authors

Thorsten Holz - t.holz@miscmag.com

Frédéric Raynal - f.raynal@miscmag.com; frederic.raynal@eads.net

References

[ref 1] The Underhanded C Contest (UCC)

<http://www.brainhz.com/underhanded/>

[ref 2] S. Lefranc, D. BÉnichou

Introduction to network self-defense: technical and judicial issues

Journal of Computer Virology, Springer

Vol.1, issue 1-2, p. 24-32, Nov. 2005

[ref 3] [Full-Disclosure] IFH-ADV-31339 Exploitable Buffer Overflow in gv

<http://www.security-express.com/archives/fulldisclosure/2004-08/0099.html>

<http://www.security-express.com/archives/fulldisclosure/2004-08/0101.html>

[ref 4] [Full-disclosure] Undisclosed Sudo Vulnerability ?

<http://seclists.org/lists/bugtraq/2005/Jul/0523.html>

[ref 5] Message Lab archive

[http://www.messagelabs.com/portal/server.pt/gateway/PTARGS_0_5882_476_319_-319_43/\[...\]](http://www.messagelabs.com/portal/server.pt/gateway/PTARGS_0_5882_476_319_-319_43/[...])

[ref 6] Sasser Worm ftpd Remote Buffer Overflow Exploit (port 5554)

<http://www.securiteam.com/exploits/5APOJOACUM.html>

<http://www.linklogger.com/TCP5554.htm>

<http://www.frsirt.com/exploits/05102004.sasserftpd.c.php>

<http://www.lurhq.com/dabber.html>

[ref 7] This is not a viruswar, this is a botwar!

<http://www.f-secure.com/weblog/archives/archive-082005.html#00000631>

[ref 8] Fighting Internet Worms With Honey pots - L. Oudot

<http://www.securityfocus.com/infocus/1740>

[ref 9] Scapy - P. Biondi

<http://www.secdev.org/projects/scapy/>

[ref 10] KARMA Wireless Client Security Assessment Tools

Dino A. Dai Zovi, Shane Macaulay ktwo

<http://www.theta44.org/karma/index.html>

[ref 11] Linux on-the-fly kernel patching without LKM

<http://www.phrack.org/phrack/58/p58-0x07>

[ref 12] SucKIT rootkit source code

<http://packetstormsecurity.org/>

[ref 13] Analyse d'un binaire SucKIT V2, Samuel Dralet
<http://forensics-dev.blogspot.com/2005/11/suckit-v2.html>

[ref 14] fakebust
<http://lcamtuf.coredump.cx/soft/fakebust.tgz>

Disclaimer

[[disclaimer](#)] The SecurityFocus editorial team supports the positive nature of this article and a legal defense of information systems, abiding by the ethical Google mantra of "Do No Evil." However, the opinions expressed in this article are not necessarily those of SecurityFocus or its owner, Symantec Corporation.

© Copyright 2006, SecurityFocus.

[Privacy Statement](#)

Copyright 2006, SecurityFocus