

Know Your Enemy: Containing Conficker

To Tame A Malware

The HoneyNet Project

<http://honeynet.org>

Felix Leder, Tillmann Werner

Last Modified: 7th April 2009 (rev2)

The Conficker worm has infected several million computers since it first started spreading in late 2008 but attempts to mitigate Conficker have not yet proved very successful. In this paper we present several potential methods to repel Conficker. The approaches presented take advantage of the way Conficker patches infected systems, which can be used to remotely detect a compromised system. Furthermore, we demonstrate various methods to detect and remove Conficker locally and a potential vaccination tool is presented. Finally, the domain name generation mechanism for all three Conficker variants is discussed in detail and an overview of the potential for upcoming domain collisions in version .C is provided. Tools for all the ideas presented here are freely available for download from [9], including source code.

1. INTRODUCTION

The big years of wide-area network spreading worms were 2003 and 2004, the years of Blaster [1] and Sasser [2]. About four years later, in late 2008, we witnessed a similar worm that exploits the MS08-067 server service vulnerability in Windows [3]: *Conficker*. Like its forerunners, Conficker exploits a stack corruption vulnerability to introduce and execute shellcode on affected Windows systems, download a copy of itself, infect the host and continue spreading. SRI has published an excellent and detailed analysis of the malware [4]. The scope of this paper is different: we propose ideas on how to identify, mitigate and remove Conficker bots.

This paper contains information about detecting and removing Conficker – either remotely, by the way it patches the Windows server service vulnerability by which it spreads, or by identifying the malware locally and wiping it from memory and hard drive. In addition to just describing how this can be achieved, we have also developed software for all the mitigation methods described in this paper. All these tools are licensed under the GPL and therefore released including source. Downloads are available from <http://iv.cs.uni-bonn.de/conficker> [9].

This paper is structured as follows: The rest of section 1 gives a very brief introduction about how Conficker infects a system and how its exploits for the Windows MS08-067 vulnerability are designed to operate. Section 2 explains the way Conficker dynamically patches the Windows server service vulnerability following successful infection. Section 3 explains how the decentralized update procedure of Conficker.B works, how the signatures in Conficker.A, .B, and .C are verified and why it is difficult to attack this procedure. Section 4 describes the hooking mechanism Conficker uses to prevent infections using this attack vector. Section 5 explains how the investigation helped to create a network scanner for infected machines. Besides actively scanning for infected machines, infections can be enumerated passively. The generation and usage of specific IDS patterns from Conficker's shellcode to detect infections is presented in section 6. Section

7 describes the domain name generation mechanism that Conficker-infected hosts are using to check for updates on a regular basis and discusses a potential solution to the aforementioned problem based on this knowledge. Further, it illustrates related issues in the pseudo random number generator based on a re-implementation in C. In section 8 we provide some details about the mechanisms Conficker implements to complicate enumeration. Conficker variants .B and .C contain blacklists of some IP addresses used by various antivirus and security companies and prevents connections to these networks. We provide some details about the network ranges contained in those blacklists and discuss how these blacklists were created. On infected systems it is important to disable Conficker as quickly as possible, to prevent any attempted countermeasures against disinfection tools. Section 9 explains our disinfection tool and demonstrates the method it uses to terminate and wipe Conficker from memory on infected hosts, while keeping the infected system's services running. Various organizations have reported re-infection after rebooting cleansed Conficker systems, so we have created a vaccination tool that prevents infection by Conficker variants .A, .B, and .C. In Section 10 we explain in detail how mutexes are used in Conficker and how we exploit the use of mutexes to create our Nonficker Vaxination tool. Besides attacking Conficker in memory and using mutexes for vaccination, it is also possible to remove the binary file Conficker installs. Although there have been reports that Conficker files have random names, this is not completely accurate. In section 11 we explain the name generation mechanism and installation path of the Conficker.B and .C DLLs. As the name generation mechanism is deterministic, this information can be used to find and remove the malicious files. In section 12, we show the impact of Conficker.C's modified domain name generation mechanism and provide information about the potential for collisions with existing domain names that Conficker.C will attempt to contact in April 2009. In section 13, we attempt to derive information about the designers and developers of Conficker, based on our findings and observations to date. We conclude our work in section 14.

The original paper included a detailed explanation about issues in Conficker, which allow exploitation. The Conficker Working Group (CWG) has requested to not include this section in this public version for various reasons. The full paper will be published in the near future.

The tools discussed in all of this paper are all licensed under the GPL and everything presented here is freely available for download from [9], including the source code.

1.1 CONFICKER INFECTION PROCESS

Conficker is delivered as a Dynamic Link Library (DLL), so it cannot run as a standalone program and must be loaded by another application. A vulnerable Windows system is generally infected with the Conficker worm via the MS08-067 vulnerability, using exploit shellcode that injects the DLL into the running Windows server service. Other possible infection vectors are accessing network shares or USB drives where the malicious DLL is started via the `rundll32.exe` application. Once infected, Conficker installs itself as a Windows service to survive reboots. It then computes domain names using a time-seeded random domain name generator and attempts to resolve these addresses. Each resolved address is contacted and a HTTP download is attempted. No successful HTTP download was witnessed until the middle of March 2009, at which point security experts observed nodes that downloaded encrypted binaries from some of the randomly generated domains.

Thinking about ways to attack Conficker's infrastructure, this DNS based update feature is obviously a potential target. However, Conficker uses RSA signatures to validate the downloads and rejects them if the check fails, and attacking RSA is not feasible.

1.2 CONFICKER'S SERVER SERVICE EXPLOIT

The Windows server service vulnerability allows Conficker to act as autonomously spreading malware, which is probably the main reason for it's success. This section describes how the vulnerable function is exploited to execute commands on the victim host. The exploit vector is a remote procedure call to the Windows API function `NetpwPathCanonicalize()`, exported by `netapi32.dll` over an established server message block (SMB) session on TCP port 445. This function takes a single argument, a path string, and canonicalizes it, e.g., `aaa\bbb\..\ccc` becomes `aaa\ccc`. However, the routine that shrinks the string

contains a security bug: by invoking a specially crafted path string, it is possible to move beyond the start of a stack buffer and control the value of a function's return address (note that this is not a classical buffer overflow where you write beyond the buffer's end). With this knowledge standard exploitation actions can be performed. Alexander Sotirov has decompiled the relevant function and published some C code with comments that greatly explain the problem [5].

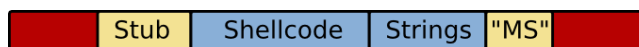


Figure 1: Conficker's shellcode and its structure in a path string

Conficker uses some standard PEB shellcode to load libraries and resolve function names. It is encoded using a one byte XOR key to prevent 0-bytes, which would be interpreted as string terminators. As displayed in figure 1, the first few shellcode instructions form a decryptor stub that, upon execution, reconstructs the original shellcode in memory. The shellcode itself loads `urlmon.dll` and locates `URLDownloadToFile()` in it, which is a function designed to download a file from a webserver and store it on the local hard drive. Both the DLL and function name are appended to the shellcode instructions and also covered by the XOR encoding. The last two encoded MS bytes are used as a stop pattern to the decryptor.

Figure 2 shows a shellcode example that was caught in a honeypot in its encoded and decoded form. As the XOR key is always `0xC4`, collecting Conficker samples is fairly straight forward: it is sufficient to XOR the whole exploit string (even with the SMB protocol information), extract the URL and download the file. These steps are easy to automate on a honeypot itself. Nevertheless, there are some additional factors to consider. We observed that downloading samples using `wget` or similar command line tools would not work for variants later than Conficker.A, even when using a spoofed user agent string. This is because the server would not submit the binary but instead returns a stream of lowercase characters. We haven't looked into the corresponding code parts yet, but there appears to be some kind of browser identification functionality within Conficker that goes beyond simple user agent string comparison. The solution is to behave exactly like an exploited system, so we wrote our own small downloader that uses `URLDownloadToFile()`, cross-compiled it on Linux and ran it in `wine` to retrieve the samples.

```

e8 ff ff ff ff c2 5f 8d 4f 10 80 31 c4 41 66 81 |....._O..l.Af.| 2c 3b 3b 3b 3b 06 9b 49 8b d4 44 f5 00 85 a2 45 |,;,;,;.I..D....E|
39 4d 53 75 f5 38 ae c6 9d a0 4f 85 ea 4f 84 c8 |9MSu.8.....O..O..| fd 89 97 b1 31 fc 6a 02 59 64 8b 41 2e 8b 40 0c |...l.j.Yd.A..@.|
4f 84 d8 4f c4 4f 9c cc 49 73 65 c4 c4 c4 2c ed |O..O.O..Ise.....| 8b 40 1c 8b 00 8b 58 08 8d b7 a1 00 00 e8 29 |.@....X.....)|
c4 c4 c4 94 26 3c 4f 38 92 3b d3 57 47 02 c3 2c |...&<O8.;.WG...| 00 00 00 50 e2 f8 8b fc 56 ff 17 93 83 c6 07 e8 |...P...V.....)|
dc c4 c4 c4 f7 16 96 96 4f 08 a2 03 c5 bc ea 95 |.....O.....| 18 00 00 00 33 d2 52 52 8b cc 66 c7 01 78 2e 51 |...3.RR..f..x.Q|
3b b3 c0 96 96 95 92 96 3b f3 3b 24 69 95 92 51 |;.....;.;$.i..Q| ff 77 04 52 52 51 56 52 ff 37 ff e0 ad 51 56 95 |.w.RRQVR.7...QV.|
4f 8f f8 4f 88 cf bc c7 0f f7 32 49 d0 77 c7 95 |O..O.....2I.w...| 8b 4b 3c 8b 4c 0b 78 03 cb 33 f6 8d 14 b3 03 51 |.K<.L.x..3.....Q|
e4 4f d6 c7 17 cb c4 04 cb 7b 04 05 04 c3 f6 c6 |.O.....{.....| 20 8b 12 03 d3 0f 00 c0 0f bf c0 c1 c0 07 32 02 |.....2..|
86 44 fe c4 b1 31 ff 01 b0 c2 82 ff b5 dc b6 1f |.D..l.....| 42 80 3a 00 75 f5 3b c5 74 06 46 3b 71 18 72 db |B...u.;.t.F;q.r.|
4f 95 e0 c7 17 cb 73 d0 b6 4f 85 d8 c7 07 4f c0 |O.....s..O.....O..| 8b 51 24 03 d3 0f b7 14 72 8b 41 1c 03 c3 8b 04 |.Q$.r.A.....|
54 c7 07 9a 9d 07 a4 66 4e b2 e2 44 68 0c b1 b6 |T.....fN..Dh...| 90 03 c3 5e 59 c3 60 a2 8a 76 26 80 ac c8 75 72 |...^Y..`..v&...ur|
a8 a9 ab aa c4 5d e7 99 1d ac b0 b0 b4 fe eb eb |.....]| 6c 6d 6f 6e 00 99 23 5d d9 68 74 74 70 3a 2f 2f |lmon..#.http://|
fd f5 ea f5 ea f6 f0 f7 ea f6 f4 f0 fe fc f4 eb |.....MS| 39 31 2e 31 2e 32 34 33 2e 32 30 34 3a 38 30 2f |91.1.243.204:80/|
a9 a5 b1 a8 c4 4d 53 |.....MS| 6d 61 75 6c 00 89 97 |maul...|

```

Figure 2: Encoded and decoded shellcode sample

One interesting aspect is a slight change in the shellcode that was observed in recorded exploits. It was probably introduced by Conficker version .B (see the next section for a discussion on versions and names): One of the first decoded assembly instructions is the `SLDT` instruction (Store Local Descriptor Table) which is widely used in malware for virtual machine detection. Because the result is not used, we assume that the only purpose of this additional instruction is to evade analysis. When this was first observed, the `SLDT` instruction was not implemented in the `libemu`-based `sctest` utility [16], and analyzing the shellcode with `libemu` was therefore not possible. However, adding the `SLDT` instruction to `libemu` was fairly easy, which means that this evasion mechanism no longer evades `sctest`. The SRI technical report contains an example of `sctest`'s output for Conficker's shellcode [4] that was generated after the `SLDT` instruction was added to `libemu`.

1.3 NAMING

The different sample naming schemes used by different antivirus companies often leads to malware naming confusion. For Conficker, the most prominent naming scheme is appending a letter so that .A indicates the first version, .B the second and so on. In this paper, we distinguish between different Conficker versions based on the mutexes they create. Each Conficker version installs a couple of named mutexes during startup, to make sure that older version of the code are not run. This is achieved by registering all previous mutex names plus an additional mutex with a different name in each version. If mutex creation fails, this indicates that another Conficker version is already running which is at least as recent as the one currently being executed. We observed three different mutex installation routines, corresponding to three Conficker variants that we call .A, .B, and .C. Although some researchers refer to versions .D or .B++, our naming convention matches the most common agreement. However, while Conficker is definitely a sophisticated piece of malware, there seems to be a flaw in its mutex generation mechanism. We assume that the Conficker authors made a mistake that effectively renders the concept of using mutual exclusion useless. We will discuss the mutexes in more detail in section 10.1.

2. THE `NetpwPathCanonicalize()` HOOK

It is quite common in modern malware to patch a vulnerability after successful exploitation, to prevent other malware from also infecting the compromised system. Conficker attempts the same approach by hooking itself into the vulnerable function `NetpwPathCanonicalize()` and intercepting corresponding incoming remote procedure calls. This hook replaces the first couple of bytes of the function's instructions with a `JMP` instruction to redirect execution to Conficker's own code. The purpose of the hook will be explained later. The following figure compares the first instructions of the original function to the hooked one (taken from an English Windows XP SP2 without any updates installed and infected with Conficker.C):

5B86A259	8BFF	MOV EDI,EDI	5B86A259 E9 A0B028A6	JMP 01AF52FE
5B86A25B	55	PUSH EBP		
5B86A25C	8BEC	MOV EBP,ESP		
5B86A25E	53	PUSH EBX	5B86A25E 53	PUSH EBX
5B86A25F	8B5D 14	MOV EBX,DWORD PTR SS:[EBP+14]	5B86A25F 8B5D 14	MOV EBX,DWORD PTR SS:[EBP+14]
5B86A262	56	PUSH ESI	5B86A262 56	PUSH ESI
5B86A263	57	PUSH EDI	5B86A263 57	PUSH EDI
5B86A264	33FF	XOR EDI,EDI	5B86A264 33FF	XOR EDI,EDI
5B86A266	3BDF	CMP EBX,EDI	5B86A266 3BDF	CMP EBX,EDI
5B86A268	0F85 8EDE0000	JNZ NETAPI32.5B8780FC	5B86A268 0F85 8EDE0000	JNZ NETAPI32.5B8780FC

Figure 3: *Unpatched and patched version of `NetpwPathCanonicalize()`*

As we can see, the `JMP` instruction takes 5 bytes (1 byte for the opcode plus a 4 bytes address), so only the first 5 bytes of the original function are modified. Conficker needs to know where the next unmodified instruction starts in order to save all (partially) overwritten instructions in a different location, otherwise execution could land in the middle of an instruction and cause errors. It has to ensure that all overwritten instructions are completely copied and that execution is resumed at the next unmodified original instruction. This is achieved by parsing the instructions with *mlde32*, a *Micro Length-Disassembler Engine 32* [6], that is included in the code. This disassembler is called in a loop, and the length values are added up until the identified instructions provide enough room for the patch (e.g., the `JMP` as in figure 3). In fact, the hook installer routine is implemented generically to work with arbitrary target functions and hooking instructions.

Conficker.C hooks a couple of other functions in different DLLs using the same method. Table 1 contains a list that we extracted from one of the samples. Which of these functions are hooked depends on the way the library was loaded. For instance, if the DLL was injected into `svchost.exe` by the server service vulnerability shellcode, it only hooks `NetpwPathCanonicalize()`. If an infected system is rebooted, the `svchost` process that provides a DNS caching service to other applications is also hooked too. The purpose of these hooks is to filter out name resolution attempts for a list of antivirus and security vendor sites whose names are embedded in the malware. You can easily verify this effect by attempting to load a web page for one such domain in a web browser. If the name resolution fails in the browser but a command line `nslookup` works for the target domain, the Conficker hook is probably active. For now, a more detailed analysis of the hooking routines is beyond the scope of this paper, but may be added in later versions.

DLL	Function
dnsapi.dll	DnsQuery_A
	DnsQuery_UTF8
	DnsQuery_W
	Query_Main
netapi32.dll	NetpwPathCanonicalize
ntdll.dll	NtQueryInformationProcess
wininet.dll	InetnetGetConnectedState
ws2_32.dll	sendto

Table 1: Functions hooked by Conficker.C

3. HOOKING INTO THE DECENTRALIZED UPDATE PROCESS

Conficker.B contains a routine to update itself by scanning incoming exploitation attempts from other infected machines and downloading the new malware binaries from the attacker. If the `..\` pattern was found in the path argument to a `NetpwPathCanonicalize()` request, it is not only blocked. Instead, the path is compared against the shellcode template to check whether the request belongs to an exploit sent by another Conficker infected computer. This is possible because of the static shellcode design. Figure 4 shows the decryption loop, a simple bitwise XOR with the constant key `0xC4` (the first box). After that, the routine tries to locate an occurrence of the substring `http://`, which, in case of a valid exploit, would point to a sample download URL. If found, the download is performed and the running Conficker code is updated on the fly to the newer version.

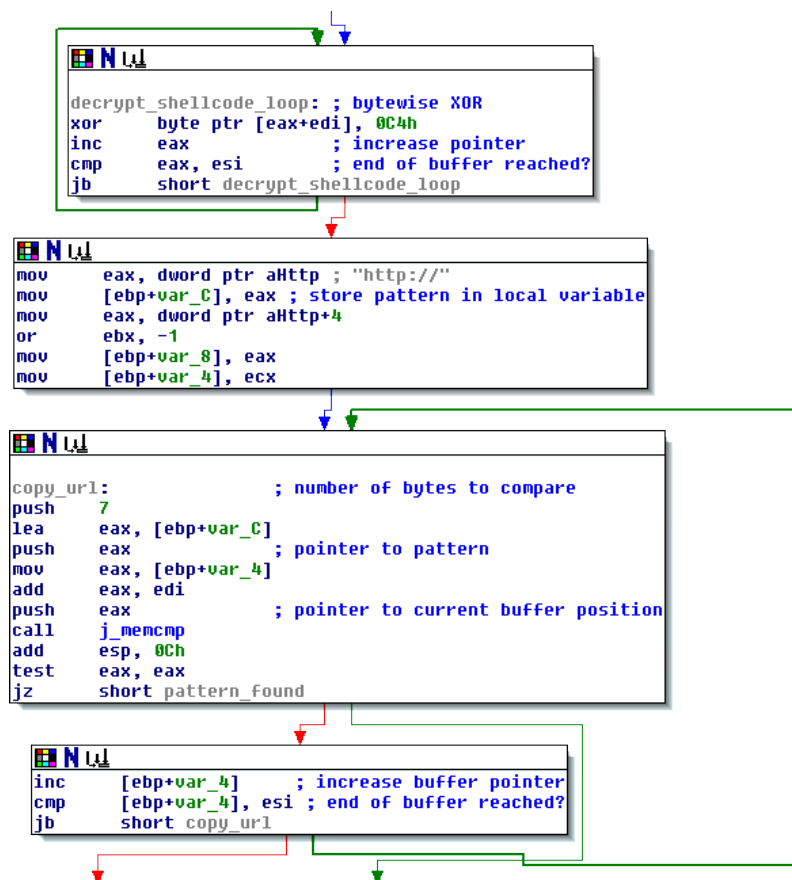
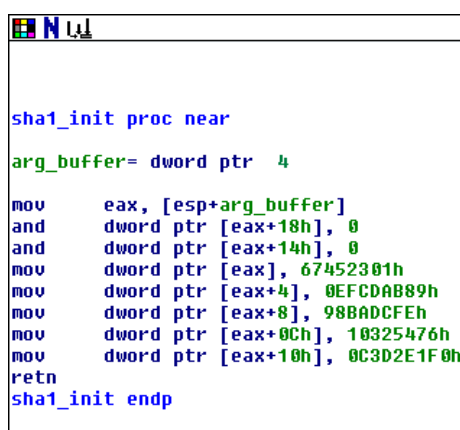


Figure 4: Shellcode decryptor and URL finder

Hooking into this update procedure requires identification of infected computers. There are several ways to do this, the most promising approach being based on DNS sinkholing, which will be discussed further in section 7. Another possible approach involves using honeypots to catch Conficker exploitation attempts, which are a strong indication that the attacking machine is itself infected. One could then send a specially crafted exploit to this machine that would trigger the download and execution of a cleaning DLL. An alternative approach is the development of a network based scanner, which we detail in the next section.

However, the Conficker authors have taken preventive measures against such an approach: a downloaded file has to carry a signature (an RSA encrypted hash) that is checked against a public RSA key. If no valid signature is found, the downloaded file is discarded. Conficker.A uses SHA1 to hash binaries and a 1024 bit RSA key. Later Conficker variants use 4096 bit RSA and a different hashing scheme which we so far haven't had time to look at in detail. This signature check is performed whenever Conficker downloads an update, no matter if it was received from a central location or another machine. Using RSA with sufficiently long keys to authenticate files makes it basically infeasible to inject other files into the update process, so we did not concentrate on attacking Conficker in this way any further. However, it might be noteworthy that we haven't seen the ability to parse incoming shellcodes in Conficker.A and .C variants, which means that, to date, the only updates possible are from version .B to .C. We were able to match the signature algorithm in Conficker.A to OpenSSL's SHA1 implementation [7]. The types, number of arguments and library calls as well as dependencies match the file hashing functions exactly to the `SHA_Init()`, `SHA_Update()`, and `SHA_Finalize()` functions of the OpenSSL library. Figure 5 shows the init function. Looking at the constant values proves it as being SHA1.



```

sha1_init proc near
arg_buffer= dword ptr 4
mov     eax, [esp+arg_buffer]
and     dword ptr [eax+18h], 0
and     dword ptr [eax+14h], 0
mov     dword ptr [eax], 67452301h
mov     dword ptr [eax+4], 0EFCDB89h
mov     dword ptr [eax+8], 98BADCFEh
mov     dword ptr [eax+0Ch], 10325476h
mov     dword ptr [eax+10h], 0C3D2E1F0h
retn
sha1_init endp

```

Figure 5: SHA1_INIT function in Conficker.A

In case of a Conficker.A update, the SHA1 hash of the binary is padded by prepending 60 "0xFF" bytes. The overall 80 bytes are encrypted using the private key and appended to the original binary. The clients verify this signature by decrypting the hash and comparing it to their own SHA1 sum of the binary. The binary is kept in memory until the verification was successful. Attacking this update procedure requires us to either break the RSA key or to create a SHA1 collision with an existing update. Even though keys with 1039 bits have been factorized at the University of Bonn [8], this cannot be performed within acceptable timescales, and Conficker.B and .C use much longer RSA keys. Neither is it feasible to attack using SHA1 collisions, because no one has observed a Conficker.A update including a hash to date (the observed downloads were from .B domains). Even if a hash for a valid update was available, a SHA1 second preimage attack (the task to compute a second input for a given hash) where the hashed data is a valid executable is still virtually impossible. We think that the way updates are signed and verified is too hard to attack, if not impossible. So we focused on the alternative methods we cover in the following sections.

4. DISSECTING CONFICKER'S PATCH

It is of general belief that Conficker patches compromised systems after infection to prevent future attacks against the Windows server service bug from being successful. However, we analyzed the relevant Conficker code elements and found that this to be not necessarily correct. Conficker does not patch the MS08-067 vulnerability. Instead, calls to `NetpwPathCanonicalize()` are analyzed by the aforementioned hooking routine to check whether they contain indications for exploits. By taking a look at the basic block graph in figure 6 it is easy to see what this function does: The first block just checks whether a string was passed and returns immediately in case of a NULL pointer. Otherwise, it moves on into the second block, where the path is scanned for the pattern `\\.\\` by calling the `wcsstr()` function. If the pattern was not found, execution follows the red arrow to a box that performs a length check: If the path string is longer than 200 wide characters, the program branches to the left block, resulting in a return value of 0. Otherwise, 1 is returned. To sum up, this function checks whether the requested path contains the unicode sequence `\\.\\` or if it is longer than 200 wchars. Depending on the return value, the calling function would skip or call the original `NetpwPathCanonicalize()` function. Conficker.C extends this approach slightly: in case of a reject, `SetLastError()` is called to simulate the processing of an invalid argument, indicated by error code 87 (`ERROR_INVALID_PARAMETER`).

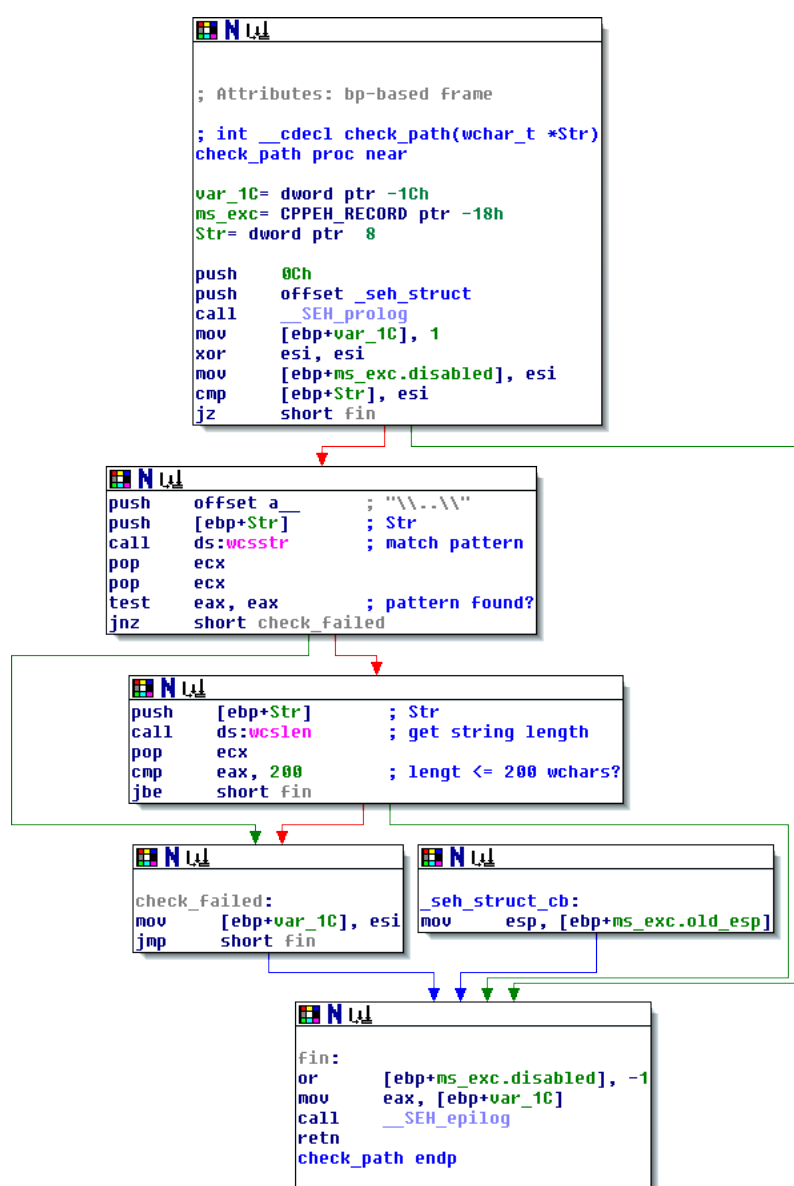


Figure 6: The filter blocking paths longer than 200 wchars or containing the pattern `\\.\\`

5. NETWORK WIDE CONFICKER SCANNING

As explained in section 4, Conficker installs a handler that checks for suspicious paths before passing it to the possibly vulnerable `NetpwwPathCanonicalize()` function. All of the three considered Conficker variants return the error code for "invalid parameters" (87) in case they either find a `\\.\\` in the path or if the path is longer than 200 wide characters. This legitimate error code is returned to the calling RPC program. The constant return code from Conficker can be exploited to remotely identify infected machines. We have developed a tool that allows system administrators to quickly and easily scan their networks for infected hosts. The tool can be downloaded from [9].

The use of `\\.\\` sequences in RPC calls to `NetpwwPathCanonicalize()` is legitimate and is evaluated by canonicalizing the path in uninfected systems. In case of valid path names, a successful canonicalization is signalled by setting the error code to `WERR_OK` (0). Besides, specific errors can be triggered by creating specially crafted paths. In case such a path name contains `\\.\\` sequences or is longer than 200 wide chars, the error code is set to the constant 87 by the handler and not by `NetpwwPathCanonicalize()` itself. As this results in different return values for both valid and specifically crafted paths, it can be used as an indication whether a host is infected or not.

Conficker installs this handler on all systems even if the MS08-067 vulnerability has been patched by the user or administrator. Thus, it is possible to identify all machines infected by Conficker and not only unpatched ones. The latter are usually machines that have been infected by other vectors, like USB-sticks or network shares.

6. INTRUSION DETECTION

There are several ways to detect Conficker's attempts to abuse the server service vulnerability in the network stream. Because of the nature of the exploit we chose to create signatures for matching against the shellcode pattern rather than against the vulnerability trigger (a path containing a `\\.\\.\\` unicode sequence), for which signatures are already publicly available. As mentioned before, Conficker uses a static shellcode with a fixed XOR key to make sure other versions can successfully decode and parse it. This has the advantage that the static string is well suited as a signature. We collected several exploit traces and processed them with *nebula* [17], an automated intrusion signature generator, to create rules for use in the *snort* intrusion detection system [18]. Based on the signatures produced, the following rules were created:

```
alert tcp any any -> $HOME_NET 445 (msg: "conficker.a shellcode"; content: "|e8 ff ff ff ff c1|^|8d|
N|10 80|1|c4|Af|81|9EPu|f5 ae c6 9d a0|0|85 ea|0|84 c8|0|84 d8|0|c4|0|9c cc|IrX|c4 c4 c4|,|ed c4 c4
c4 c4 94|&<08|92|\;|d3|WG|02 c3|,|dc c4 c4 c4 f7 16 96 96|0|08 a2 03 c5 bc ea 95|\;|b3 c0 96 96 95 92
96|\;|f3|\;|24|i| 95 92|Q0|8f f8|0|88 cf bc c7 0f f7|2I|d0|w|c7 95 e4|0|d6 c7 17 f7 04 05 04 c3 f6
c6 86|D|fe c4 b1|1|ff 01 b0 c2 82 ff b5 dc b6 1b|0|95 e0 c7 17 cb|s|d0 b6|0|85 d8 c7 07|0|c0|T|c7 07
9a 9d 07 a4|fN|b2 e2|Dh|0c b1 b6 a8 a9 ab aa c4|}|e7 99 1d ac b0 b0 b4 fe eb eb|"; sid: 2000001;
rev: 1;)
```

```
alert tcp any any -> $HOME_NET 445 (msg: "conficker.b shellcode"; content: "|e8 ff ff ff ff c2|_|8d|
O|10 80|1|c4|Af|81|9MSu|f5|8|ae c6 9d a0|0|85 ea|0|84 c8|0|84 d8|0|c4|0|9c cc|Ise|c4 c4 c4|,|ed c4
c4 c4 94|&<08|92|\;|d3|WG|02 c3|,|dc c4 c4 c4 f7 16 96 96|0|08 a2 03 c5 bc ea 95|\;|b3 c0 96 96 95
92 96|\;|f3|\;|24|i|95 92|Q0|8f f8|0|88 cf bc c7 0f f7|2I|d0|w|c7 95 e4|0|d6 c7 17 cb c4 04 cb|{|04
05 04 c3 f6 c6 86|D|fe c4 b1|1|ff 01 b0 c2 82 ff b5 dc b6 1f|0|95 e0 c7 17 cb|s|d0 b6|0|85 d8 c7 07|
O|c0|T|c7 07 9a 9d 07 a4|fN|b2 e2|Dh|0c b1 b6 a8 a9 ab aa c4|}|e7 99 1d ac b0 b0 b4 fe eb eb|"; sid:
2000002; rev: 1;)
```

The first of the above rules covers the full static part of Conficker.A's shellcode. (c.f. figure 2). It starts with a `CALL` instruction which is part of the decryptor stub's `GetPC` sequence and ends with `ac b0 b0 b4 fe eb eb`, which decodes to `http://`. The next part would be an IP address that could vary and is therefore not covered anymore. The second rule matches the shellcode of exploits sent by Conficker.B. It is easy to see that there are only few differences which are mainly related to the additional `SLDT` instruction. It is quite obvious that both rules can be generalized into one that matches all Conficker versions but then the information about the attacking version would not be present anymore. Below is a real world example of a rule match for Conficker.B:


```

[**] [1:2000002:1] conficker.b shellcode [**]
[Priority: 0]
03/29-23:43:22.095678 88.29.81.25:2238 -> 127.0.208.64:445
TCP TTL:114 TOS:0x0 ID:11608 IpLen:20 DgmLen:832 DF
***AP*** Seq: 0x594AF9FF Ack: 0x4657E1BC Win: 0x4136 TcpLen: 20

```

As both signatures consist of contiguous byte sequences, pattern matching is straightforward and it is questionable whether a complex IDS like snort is really necessary to detect them. We have used the lightweight *ngrep* as a minimal Conficker IDS and found it to be well suited. Here is how the tool should be started for detecting the version .B shellcode pattern:

```

$ sudo ngrep -qd eth0 -W single -s 900 -X
0xe8ffffffc25f8d4f108031c4416681394d5375f538aec69da04f85ea4f84c84f84d84fc44f9ccc497365c4c4c2cedc4
c4c494263c4f38923bd3574702c32cdcc4c4c4f71696964f08a203c5bcea953bb3c096969592963bf33b24699592514f8ff8
4f88cfbcc70ff73249d077c795e44fd6c717cbc404cb7b040504c3f6c68644fec4b131ff01b0c282ffb5dcb61f4f95e0c717
cb73d0b64f85d8c7074fc054c7079a9d07a4664eb2e244680cb1b6a8a9abaac45de7991dacb0b0b4feebeb 'tcp port 445
and dst net 127.0.208.0/24'

```

As far as we observed the shellcode can always be found within the first 832 bytes of the packet payload. *ngrep* is started with the option to investigate the first 900 bytes to be on the safe side but to skip the rest for performance reasons. It then spits out a line as displayed below for each match. A matching for Conficker.A shellcodes can be performed accordingly.

```

T 85.178.36.87:4120 -> 127.0.208.89:445 [AP] .....SMB%.....l.....
...T...T...&...@...\.P.I.P.E.\.....H.H.D.H.H...l.....1...\.
EvhpfTtbTUyghlXpIyPnXXBvxxkIOONuVUyJXtrZdGHBSFggPPcwUYwgzFzeSxPGWvLnLqQfKfSkaxfYnrSDUunJctDgxwIeYhvc
....._O..l.Af.9MSu.8....O..O..O..O..Ise....&<O8.;.WG...O.....;.....;i..QO..O.
.....2I.w...O.....D...l.....O.....s..O...O.T.....fN..Dh.....].....
.....MskbeUNUPFXJIBBWGrpRzQeGZcUQvgpaZrKtXBQxPUAMHXxRnQgKHBeltTSYFuPVpDFukGRpBrJqqbOiAtP
YlwxPLYeYeswLJPVPTVSKWglaqyPyoPqjPABgqyTfsoubLEqNMnYCPoNPkKSmnJFLcfIAVBUDjhnjUaWQiNZYVATfMxVatUbIA\
....\.....\A.C.B.I.N.Z.K.....oCDXX'..oLENAWZGFONDZUZYQDVRDRDLZIMZGEBNGQIHQBKXXJ.J$.7.bESCXGK
MDMZ.....\.....

```

7. DOMAIN NAME GENERATION

Conficker generates a series of domain names from which it tries to download updates. Conficker.A and .B create 250 domains per day. This puts high load on the contacted domains and can easily lead to a denial of service against them. Various organizations have made efforts to attempt to pre-register these 2 x 250 daily domains in order to hinder Conficker from retrieving updates and to track infected hosts. Conficker.C tries to evade this defensive approach by creating 50.000 domains per day, making pre-registration logistically challenging. Conficker.C randomly chooses 500 out of these domains, which are then contacted for updates. The random selection of these 500 is based on Windows' random number generation. A seed is calculated based on Windows' performance counter, system time, uptime in milliseconds (tick count), thread id, and processor ticks since last boot. This creates a rather unpredictable seed and results in different seeds for every running Conficker instance. There is a random `Sleep()` period of 40 to 50 seconds between every two connections attempts. This period includes the time for DNS lookup and HTTP connection. After an unsuccessful update attempt, Conficker.C sleeps for 24 hours. In the case of a successful update, Conficker waits 4 days before continuing to attempt to download new updates. As the next domain to be contacted is chosen randomly, the load is equally distributed over many name servers but leads to the problem that there is no guaranteed set of domains that is contacted on a given day by every host, significantly increasing the effort involved in mitigation at the sinkhole or DNS registrar level.

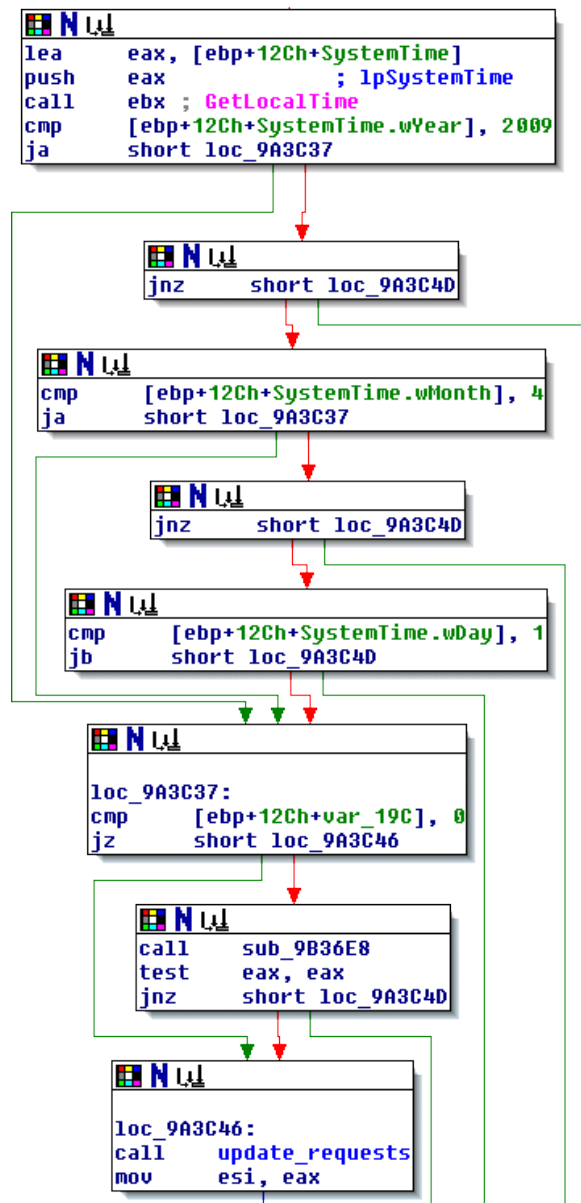


Figure 7: The C variant starts to scan for updates after April 1st, 2009 local time

As mentioned earlier, Conficker.A and .B are already using 250 domains per day to retrieve updates. Conficker.C contains code that will start to look for updates after 1. April 2009 local time, as the code flow in figure 7 shows. It is this hardcoded date value within the code that has generated such a high degree of press speculation about what the Conficker botnet will or more likely won't happen on April Fools day. Unlike the domain generation algorithm, which retrieves a GMT value from remote hosts, this new check is performed against the host's clock. Computers that have their clock set to a future time will already try to download updates. While these updates are signed based on RSA (c.f. section 3) and cannot easily be forged, it is possible to use these domains to identify infected computers. Various organizations have already registered a range of those domains and are actively sinkholing requests to facilitate monitoring the number of infections and to prevent Conficker's owners from registering them for updates. This registration of Conficker domains can also be used to identify infected nodes passively. The generated domain names can also be used by network administrators to identify infected machines by monitoring the DNS for suspicious requests. We have developed a tool that generates the domain names for Conficker variants .A, .B, and .C, and the tool and source code can be freely downloaded from [9].

Conficker employs HTTP requests for updates, which hide update requests amongst the regular web traffic patterns found in most networks. To be even more stealthy, Conficker pre-resolves the domain names and uses only plain IP addresses in the HTTP Host header. Thus, the use of application level gateways and host-based filtering of this traffic is not easily possible.

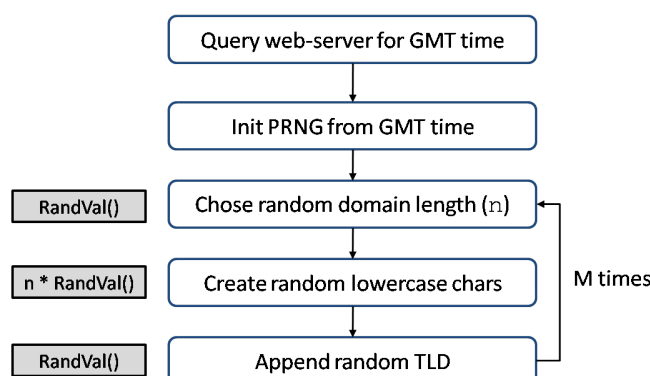


Figure 8: Domain name generation algorithm

7.1 THE DOMAIN NAME GENERATION ALGORITHM

The Domain Name Generation takes place in five steps, which are depicted in figure 8. The algorithm is identical for the three considered Conficker variants. The only difference between the three variants are the initialization vectors and the constants used in the pseudo random number generator. We have developed a tool that generates the domain names for all Conficker variants. The tool and its source code is freely available from [9].

In a first step, a public web-site is queried in order to get a response that includes the current time based on GMT. Conficker.A and .B randomly contact one of the following web-sites:

- baidu.com
- google.com
- yahoo.com
- msn.com
- ask.com
- w3.org

Conficker.C uses three more web-sites in addition to those above:

- facebook.com
- imageshack.us
- rapidshare.com

Selecting such high profile websites as these for time synchronization makes it almost impossible for system defenders to simultaneously disable all target time sources in a co-ordinated effort.

The HTTP response returned contains a Date field in the header:

```

HTTP/1.1 200 OK
Date: Fri, 20 Mar 2009 17:01:13 GMTServer: BWS/1.0
Content-Length: 1809
Content-Type: text/html
Cache-Control: private
Expires: Fri, 20 Mar 2009 17:01:13 GMT
  
```

The day, month, and year are extracted from this time stamp and are used to create a seed for Conficker's custom pseudo random number generator (PRNG). This ensures that the PRNGs of all infected hosts are started with the same seed. This result is deterministically computed on all infected hosts. The seeding is identical for Conficker.A, .B, and .C except for the constants used in its' initialization vectors:

```
SystemTimeToFileTime(time, filetime);
prng_key = filetime * CON1 / CON2 + CON3;
```

The GMT date is converted to `FileTime`, which is "the number of 100-nanosecond intervals since January 1st, 1601" [10]. Conficker combines the two resulting 32-bit values into one 64-bit value and computes the start key of the PRNG as shown above. The three constants differ among the different versions of Conficker (c.f. table 2). Thus, all hosts infected with the same variant start the PRNG with the same seed.

Constant	Conficker.A	Conficker.B	Conficker.C
Seeding			
CON1	0x463DA5676	0x352C94565	0x2682D10B7
CON2	0x058028E44000	0x058028E44000	0x19254D38000
CON3	0x0B46A7637	0x0A3596526	0x0F1E34A09
PRNG			
CON_RAND	0x64236735	0x53125624	0x4F3D859E
CON_DBL	0.737565675	6.26454564e-1	0.946270391

Table 2: Constants used in the PRNGs

All of the following steps shown in figure 8 are based on the fact that the PRNG stays in sync amongst all infected hosts. The three steps are repeated until all domain names for the given GMT date are created. First a random name length is chosen. Conficker.A and .B can generate domain names with 8 to 11 characters in length, which leads to conflicts with existing domains only in very few cases. Conficker.C creates domain names of length 4-9, which leads to 150 to 200 conflicting domains per day. We have precomputed those conflicts for April 2009. The list can be downloaded from [9], and a short overview about the colliding IP addresses and organizations can be found in section 12. After the domain length is chosen, the custom PRNG is used to create the lowercase characters for the domain name. After this base name is generated, a random top level domain (TLD) is chosen from a hardcoded list. Conficker.A uses the following TLDs:

.com, .net, .org, .info, .biz

Conficker.B employs three more addition to those:

.cc, .cn, .ws

Conficker.C uses 110 TLDs with no overlap to the Conficker.A and .B TLDs except for China ".cn". The large amount of TLDs targeted leads to the situation that many different registrars are responsible for those TLDs, in many countries around the world and attempting to block all of these domains requires significant international cooperation and coordination. The full list of Conficker.C TLDs can be seen in our "downatool2" source code [9]. For Conficker.A and .B 250 domains are generated per day. Conficker.C creates 50.000 domains per day with the implications explained above.

	Conficker.A	Conficker.B	Conficker.C
Domains/day	250	250	50.000
Domain name length	8-11	8-11	4-9
TLD suffixes	5	7	110

Table 3: Domain name generation facts

7.2 CUSTOM PRNG

All Conficker variants use the same pseudo random number generator algorithm. They only differ in two constants used. The PRNG is based on floating point operations. Our re-implementation in C is shown in the code excerpt in figure 9. In line 10, the global variable holding the current state (`prng_key`) is converted from a 64-bit integer into its floating point representation (`d2`). In line 11, the original state (as integer) is multiplied by a 32-bit constant (`CON_RAND`). Based on the results, `res1` is computed using `s_val = sin(d2)` and a second constant (`CON_DBL`) as shown in line 15. The constants vary for all considered versions of Conficker. The last mathematical operation is adding the logarithm of the floating point representation of the current state.

```

1     DWORD64 prng_key;
2
3     int RandVal(void)
4     {
5         double res1;
6         double d2;
7         double s_val;
8         DWORD64 prod;
9
10        d2 = prng_key;
11        prod = prng_key * CON_RAND;
12
13        s_val = sin(d2);
14
15        res1 = ( ( (prod + s_val) * d2) + CON_DBL ) * d2 );
16        res1+= log(d2);
17
18        *(double*)&prng_key = res1;
19
20        return prng_key;
21    }

```

Figure 9: Reimplementation of the custom PRNG

In the last step (line 18), the floating point representation is copied bit-by-bit into the integer data type. The result is the new state and return value of the PRNG. Although we are not sure if it was intended by Conficker's developers, this last step is some kind of protection for the PRNG and thus for the domain name generation. Because of this copy operation, every bit of the floating point representation is required to be identical for all PRNGs using this algorithm. If bits change, the PRNG loses sync to Conficker's implementation. While cross-compiling the algorithm using the *MinGW* Windows cross-compiler, we discovered that the cross-compiled version drifts out of sync after a few hundred operations because the `log()` function used by *MinGW* differs slightly from the implementation in the `msvcrt.dll` used by Conficker. The digit at position 10^{-13} differed because of different roundings. This resulted in a single bit that was different and brings the PRNG and therefore the domain generation out of sync.

8. CONFICKER BLACKLISTS

Conficker variants .B and .C contain blacklists of IP address ranges to prevent them from attacking and contacting hosts related to antivirus vendors (AV), some security companies, and Microsoft. We have

published the blacklists for .B and .C on our webpage [9]. The blacklist consists of network ranges with a start address and end address of each network. Conficker.A does not only contain blacklists, it also checks for reserved RFC 1918 [11] and special IP addresses, like multicast addresses. The introduction of blacklists in .B can therefore be seen as an improvement for avoiding detection from AVs and Microsoft, and evidence of the worm's author's continuing response to developments in the whitehat community. As the corporate systems typically owned by this type of organization are more likely to be fully patched against the MS08-067 exploit, this behavior may also increase spreading performance by avoiding low return netblocks. We have tried to resolve the owners of the networks contained in the blacklists. The results can be downloaded from [9]. The networks of variant B can completely be resolved using regular *whois* queries. They resolve to AVs, security companies, and Microsoft. AVs include

- Kaspersky
- Trend Micro
- Symantec
- McAfee
- F-Secure
- Avira
- Bitdefender

Different Microsoft companies include

- Microsoft Corp.
- Microsoft Education
- Microsoft License
- Microsoft Visual Studios

The networks included in Conficker.C's blacklist do not completely resolve by *whois* queries. BGP routing information does not provide much useful information about the owner of those networks, although Maxmind's [12] IP to organization lookup does provide useful information. All network ranges we have looked up using Maxmind's public service resolved to either AVs or Microsoft. A possible conclusion from this observation is that the Conficker authors may have used Maxmind's *IP address to organization* database to create the blacklist from this "insider's" information, while the blacklist in .B has been created based on *whois* information.

9. MEMORY DISINFECTION

Conficker versions have introduced more and more security checks to avoid removal. One is the detection of removal tools. In order to apply disinfection or vaccination tools, Conficker has to be terminated first, which is hard without being able to apply a removal tool. We have developed a tool that successfully terminates all running Conficker instances by wiping it from memory. It can be freely downloaded from [9].

Conficker makes use of multiple layers of packing, that differ from sample to sample. When Conficker is running, its code and data is fully unpacked in memory. This fact can be exploited to identify and terminate Conficker while keeping the system running. Our tool scans the memory allocated to all running processes, checks if they host Conficker, and terminates all Conficker threads while the regular part of the process continues to run. The general procedure is depicted in figure 10. The tool opens each running process, reads the memory of the process and performs a string search for the patterns of Conficker.A, .B, and .C. The tool and source code is freely available for download from [9].

A few people have reported slight variations in the code of different Conficker versions, such as the so called Conficker.B++. Therefore binary code representation alone does not seem ideal for creating signatures – at least not in our case, in which we only had access to a small number of samples and therefore couldn't guarantee finding a unique code signature. However, one pattern that has to exist in all variants is the RSA key that is used to check the update signatures. Without a unique key, different Conficker samples cannot verify the signature of the same update. Thus, they have to be unique. Furthermore, the RSA keys provide

long signatures of 128 bytes (1024 bit RSA) for Conficker.A and 512 bytes (4096 bit RSA) for Conficker.B and .C. As the keys have high entropy, there is a high probability that these patterns will not be found in another process. Table 4 in appendix A shows how the RSA keys are stored in memory. Note that before they are used for decryption, the byte order is reversed.

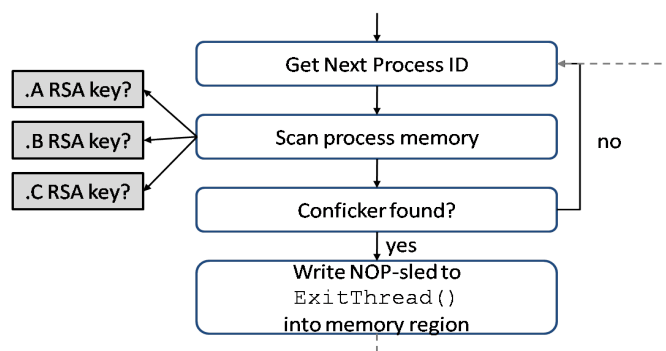


Figure 10: Disinfection process

A major barrier to easy termination is that Conficker runs inside another process. In most cases, this is a system process, such as `svchost.exe`. These processes cannot simply be terminated as this would obviously lead to system instability. To find a signature it is therefore necessary to terminate the Conficker threads without affecting other running threads. During startup, the Conficker code and data are copied into a single heap segment that resides inside the running process' memory. A new thread is started in this heap segment and the DLL is unloaded again, because `DLLMain()` returns an error. In case the signature is found in the memory of a running process, the size of the memory segment is determined and overwritten by a large number of NOP instructions (NOP sled) that end in a shellcode that calls `ExitThread()`. This way, all running Conficker threads terminate themselves and the malicious code will be wiped from memory. This includes Conficker's regular code, the MS08-067 exploit filter and each thread calling `NetpwPathCanonicalize()`, which is terminated too. This ensures that re-exploitation over the network cannot re-occur, even using the modified exploit method we described above. An adjustment of the NOP sled to pass around the filter is possible and would keep the network service running, for both friends and foes.

When Conficker is wiped from running memory, the mutexes it created (c.f. section 10) remain active inside the process and prevent a re-infection. It must be noted that this disinfection is only temporary and Conficker is reloaded after reboot. Other means, like our "*Nonficker Vaxination tool*" presented in section 10 or any other "trusted" Conficker removal tool, should therefore be applied to the system. The full disinfection tool and source code can be freely downloaded from [9]. Example output is shown below:

```

Examining [4] System: no match
Examining [380] smss.exe: no match
Examining [644] csrss.exe: no match
Examining [668] winlogon.exe: no match
Examining [712] services.exe: no match
Examining [724] lsass.exe: no match
Examining [868] svchost.exe: no match
Examining [948] svchost.exe: no match
Examining [984] svchost.exe: no match
Examining [1036] svchost.exe: no match
Examining [1092] svchost.exe: no match
Examining [1912] svchost.exe: MATCH at offset 00A152E0 of block 00A00000
Pattern for Conficker.A found
Injecting shellcode
Examining [1344] spoolsv.exe: no match
Examining [1824] alg.exe: no match
Examining [524] wuauclt.exe: no match
Examining [188] notepad.exe: no match
Examining [416] svchost.exe: no match
Examining [628] calc.exe: no match
  
```

10. NONFICKER - VACCINATION USING MUTEXES

Several organizations reported that computers which have been cleaned of Conficker infections were immediately re-infected on restart. There are several possibilities for the cause of this behavior. One is that Conficker's autostart ability and on-disk binaries were not correctly removed. Another is that the computers were immediately reinfected by other compromised computers via the (local) network. We have developed a tool that exploits the use of mutexes by Conficker to prevent such re-infection and it can also be used to identify a local infection. This can be considered a vaccination because it can be installed as a preventive feature as well as used after cleanup to prevent future re-infection. The tools for identification of local infections as well as for vaccination can be freely downloaded from [9]. Conficker creates different mutexes to ensure that only one instance of the latest version can run at any time. When these mutexes are already registered, Conficker terminates without performing malicious actions. We extracted the mutex name generation routines from Conficker.A, .B, and .C samples, re-implemented them in C and created a tool [9] that registers these mutexes. In the event that a Conficker variant is subsequently run while the tool is active, Conficker will terminate, effectively vaccinating the computer from Conficker infection.

10.1. MUTEX GENERATION

The different Conficker variants register and use different mutexes. Conficker.A computes a CRC32 checksum of a buffer containing the host name. We were able to match the CRC32 implementation inside Conficker.A to that of the *OpenSSL* library [7], based on constant structures used, function argument counter and type as well as dependencies among the functions. The full mutex is created by integrating the CRC32 checksum into the string "*Global*\[CRC32 - checksum]-7". The "*Global*" prefix is used to create a system-wide mutex. A full examples for an A mutex is *Global*\2900278491-7.

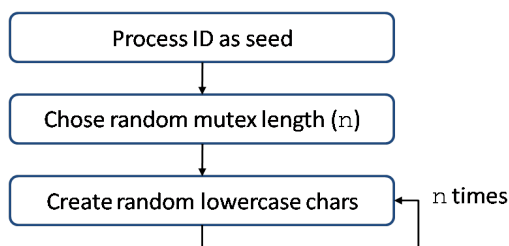


Figure 11: Local mutex generation process

Conficker.B uses two mutexes. The first mutex is local for the running process and checks if a local Conficker thread is active. If so, Conficker immediately exits. The second mutex is a global mutex that is used to check if another Conficker instance is running on the same machine. This is necessary as Conficker.B can also be run from other host executables, like e.g., *rundll32.exe*. The local mutex is based on the process ID. As depicted in figure 11, Conficker seeds the *msvcr* PRNG with it's process ID. Based on this seed, a random length of 10-16 characters is chosen and that amount of lower-case characters from the range *a-z* is generated. The global mutex is similar to the one created in Conficker.A. It is created from the host name and uses the format "*Global*\[CRC32 - checksum]-7", too. The only difference is that a different checksum computation is used. Code excerpt 16 compares our C re-implementation to a code fragment found at [13].

As this mutex is related to CRC32, as well, we assume that Conficker's authors were looking for a more lightweight implementation than that of *OpenSSL*. If both CRC32 implementations were correct, the Conficker.A and .B mutexes would conflict and not run simultaneously. But as the new checksum calculation fails to build the correct CRC32 checksum, Conficker.A and .B can run at the same machine at the same time, which was presumably not the authors' intention. We suspect that the developers didn't check if both algorithms produced the same checksum. If this second mutex is already present Conficker marks the DLL it was started from for deletion upon the next reboot using the API function `MoveFileEx(fname, NULL, MOVEFILE_DELAY_UNTIL_REBOOT)`.

Conficker.C is generating three different mutexes, a local one and two global ones. The local mutex is used to check if Conficker.C is already present in the current process. The first global mutex is used to prevent a backwards infection with Conficker.B. It is not checked if it is already present. The third mutex is used to verify that no other processes are already running version .C. Conficker terminates and deletes itself if this mutex is already present. The generation of Conficker.C's local mutex is nearly identical to that of Conficker.B except that the process ID is XOR-ed with 0x630063. The first global mutex is generated identically to Conficker.B and therefore a CRC32 variant of the host name. The second global mutex uses the same CRC32 variant of the host name but appends 0x63 in decimal (99) instead of the 7. An example for a Conficker.C mutex is "Global\1394688804 - 99". It may just be a coincidence, but 0x63 (99) is the ASCII code for "c". The same value is contained twice in the XOR key described above.

```

void CRC32Init()
{
    for(DWORD x = 255; x > 0; x--)
    {
        DWORD z = x;
        for(DWORD y = 8; y > 0; y--)
        {
            if(z & 1)
                z = (z >> 1)
                    ^ CRC_POLYNOMIAL;
            else
                z >>= 1;
        }
        dwCRCTable[x] = z;
    }
    bTable = TRUE;
    return;
}

int simplified_crc(unsigned char* name, int name_len) {
    DWORD result = 0xFFFFFFFF;
    register int i, s;

    for(i = 0; i < name_len; ++i) {
        DWORD c = name[i];
        for(s = 8; s > 0; s--)
        {
            if((c ^ result) & 1)
                result = (result >> 1)
                    ^ 0x0EDB88320;
            else
                result >>= 1;

            c >>= 1;
        }
    }
    return result;
}

```

(a) CRC32 implementation from [13]

(b) Our assembly-based C re-implementation

Figure 12: CRC32 Implementation from [13] (left) and from Conficker.B and .C (right)

10.2. USING NONFICKER

The described mutexes can be used to check if a system is infected with Conficker, as well as for registering those Mutexes to prevent (re-)infections. Microsoft describes the situation as: "If you are using a named mutex to limit your application to a single instance, a malicious user can create this mutex before you do and prevent your application from starting" [14]. We have developed a small tool that scans the system for the existence of those mutexes and warns the user. The tool tries to create the local B and C mutexes for each process ID found on the system. This reveals Conficker mutexes if the mutex was created non-locally inside a process. In the case where such a mutex exists, the process name will be displayed. In addition, the global mutexes are checked for existence as an indication of a possible Conficker infection. The tool is freely available at [9] and example output is:

```

-----
Checking for Conficker.A...WARNING: Found a possible infection
Checking for Conficker.B...clean
Checking for Conficker.C...clean
Your system is infected!
Please install the Vaxination tool and restart your computer.

```

Besides just scanning for infections, these mutexes can also be used to prevent Conficker from running. This works like a vaccination. The system checks for the presence of these particular mutexes and therefore other services that need them cannot run. For a permanent resistance against Conficker variants .A, .B, and .C, we have developed a DLL (Nonficker.dll) that can be loaded by any process. If this DLL is loaded before Conficker is executed by Windows' *svchost*, such as when the MS08-067 vulnerability is exploited, Conficker will terminate immediately, without performing any actions. To achieve this, the DLL must be registered as a

system service. This allows us to vaccinate a system with Nonficker from system startup. We have developed a setup program that installs the `Nonficker.dll` as an `svchost` service. The use of Nonficker is especially useful in situations where people have reported that systems were re-infected after a reboot. Nonficker and its source code can be freely downloaded from [9].

11. FILE REMOVAL

Many organizations have claimed that Conficker infections cannot easily be identified, because Conficker uses random file names. While this is true for Conficker.A, it is not the case for the .B and .C variants. It is true that the function `rand()` is used to create the names, but a seed is calculated that is based on the host name of the infected computer. Therefore a deterministic name is generated. Conficker.B and .C copy themselves to Windows' system directory to a DLL with the deterministically computed name. Conficker.C uses this calculation to remove Conficker.B infections from an infected system before installing itself.

We have developed a small tool that calculates the DLL name and possible installed autorun registry entries. It can be freely downloaded from [9]. After being run on the infected system, the tool can be used to manually remove the Conficker DLL. Unfortunately this isn't straightforward because Conficker attempts to hide itself. Files are protected by hiding them and setting the attributes to those of system files. Furthermore, special rights are given to those files to further restrict access to them. The Conficker DLLs are normally not visible within Windows Explorer or other tools. However, custom applications can be used to prove their existence, as shown in the following example:

```
C:\Python25>python.exe
>>> f=file("c:/windows/system32/syyisl.dll","r")

IOError: [Errno 13] Permission denied: 'c:/windows/system32/syyisl.dll'

C:\Python25>dir c:\windows\system32\syyisl.dll
Directory of c:\windows\system32

File Not Found

C:\Python25>dir /ah c:\windows\system32\syyisl.dll
Directory of C:\WINDOWS\system32

08/04/2004  01:00 PM           171,376 syyisl.dll
              1 File(s)           171,376 bytes
```

The algorithm for generating the installation file names of Conficker.B and .C is depicted in figure 13. It is a combination of existing functions used in the global mutex generation and the domain name generation. In a first step the current host name is determined and checksummed using the variant CRC32 algorithm shown in code excerpt 12. While for the mutexes, this number was used right away, here it is XOR-ed with a variant specific 32-bit value. The XOR values specific for the Conficker variant file name generation are:

- Conficker.B : 0x045419005
- Conficker.C : 0x0C7BD45E1

The resulting value is used to initialize the random number generator of the imported `msvcrt.dll` (`srand()`). As this seed depends only on the host name, it can even be computed remotely, assuming the host name is known. The rest of the calculation is similar to the domain name generation, except that the `msvcrt` PRNG (`rand()`) is used. At first the length of the file name is chosen with a length of 5-8 characters. In the last step, this number of lower case characters is generated and the suffix ".dll" appended. The Conficker DLL is then copied to this filename into the system folder.

It is to be noted that Conficker.B and .C use a similar name generation for the registry values. To the time of writing this paper, we have only extracted those keys, but want to combine all information into one single removal tool. Please visit the tool download site [9] for subsequent updates.

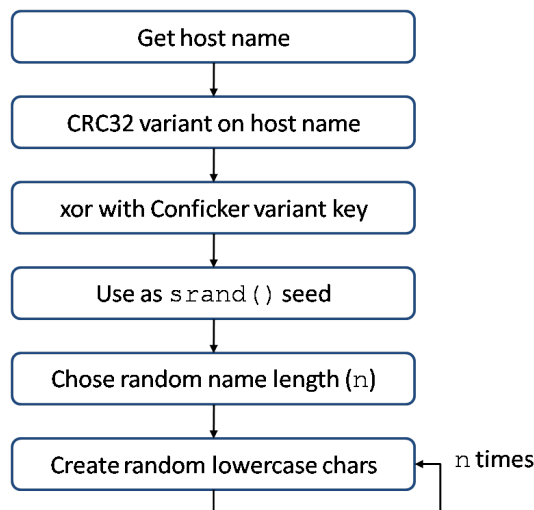


Figure 13: File name generation process for variants .B and .C

12. CONFICKER.C DOMAIN COLLISIONS FOR APRIL 2009

We have pre-computed Conficker.C domains for the updates starting on 1st April 2009. Due to the shorter names (4-9 characters), there are about 150 - 200 collisions with existing domains per day, as can be seen in figure 14 a. A list of colliding domains for April can be downloaded from [9]. The many conflicts created by Conficker.C help to hide real update domains, but there are a range of IP addresses that occur regularly, as can be seen in figure 14 b.

We have already observed a registrant for all unassigned .to, .as, .cl, and .com.mx domains that sinkholes requests to those domains. In addition, there is another sinkhole that has already been used in Conficker.B monitoring. The IP addresses of those sinkholed domains are not included in the presented results. There are still 18 IP addresses that create collisions with more than 24 domains for April 2009. Six domains collide with more than 100 domains and one with more than 200 domains. The full list of collisions can be downloaded from our webpage at [9].

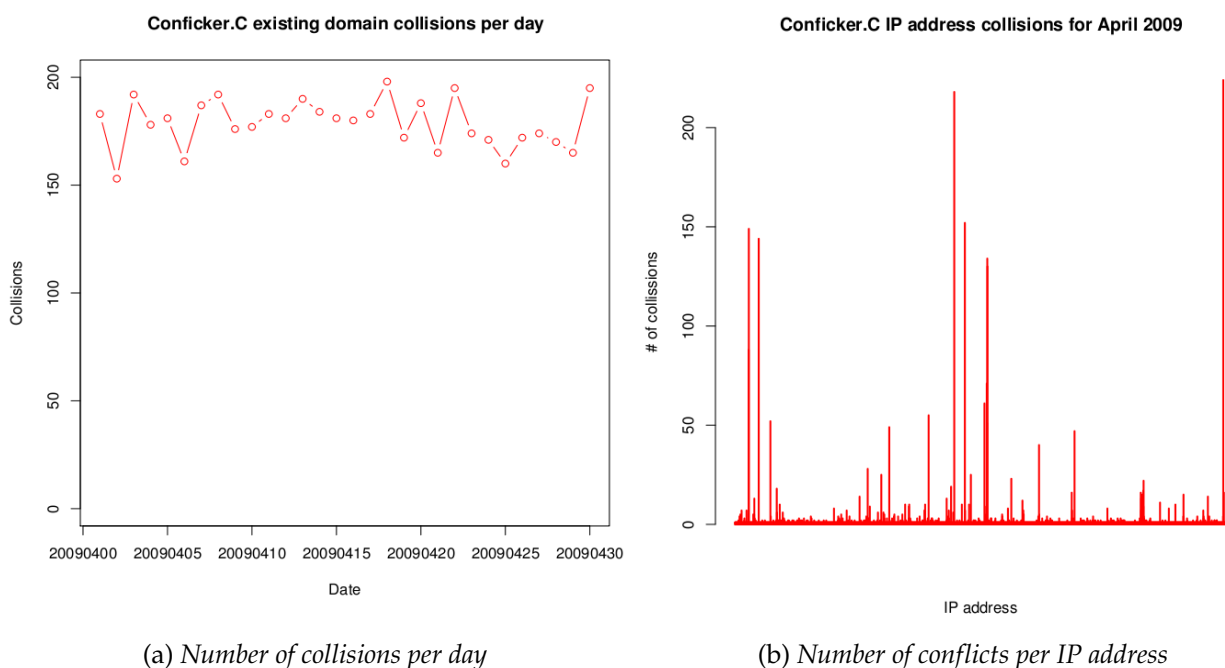


Figure 14: Collisions with Conficker.C domains for April 2009

13. WHO IS BEHIND CONFICKER?

This section is not about profiling. We won't discuss questions such as "How many authors are involved in Conficker development?", "What is their location?" or "What is their motivation?". While these questions are definitely interesting to discuss, it is beyond the scope of this technical paper. However, we can try to draw a picture from the facts presented in this paper and learn something about the people potentially behind Conficker. Most notably, the malware seems to have been developed in a rather professional manner. By this, we don't refer to the different revisions, which is quite common even for less sophisticated malware. In fact, we think it is difficult to write a program with functionality comparable to Conficker's without software design experience and a clear development concept. There must be at least some organized approach behind this worm, and we would not be surprised to find indications that the source code was organized in a revision control system. The use of mutexes ensures that only the newest version is running (c.f. section 10). Also, the coding style seems to be quite good, as far as you can tell from inspection of the assembly code. Return values are checked in most cases, error handling is performed, and functions are held generically to suit multiple tasks. Speculating on the programming language is difficult, however, there are not much indications for object orientation, so we assume Conficker was written in an imperative language.

To try and avoid speculation and only look at the facts available, it seems that whoever wrote Conficker is a capable software developer. Many security experts wonder why strong cryptography is either absent in most malware families or at least implemented incorrectly, often rendering it completely useless. It is widely known that adoption of strong encryption and even more importantly strong signatures can substantially raise the bar against defeating malware. Conficker's design does not appear to particularly care about privacy, which makes sense as analysts have complete control over a sample and can eventually snoop on its communication anyway. Instead, it relies on making use of signed updates, using a combination of sufficiently secure hashing and a private/public encryption scheme, namely RSA, with a key size that is considered secure. The length of the key was even increased from 1024 bits in Conficker.A to 4096 bits in later versions, showing that, even though the .A key would already be very hard to break, the keys were changed just to be on the safe side.

Besides correctly using strong cryptography, the authors have also been smart enough to avoid implementing the functions themselves (which is a common cause of failure in poorly written malware). Instead, they borrowed code from OpenSSL [7], which is an open source project and considered high quality and well tested. The pseudo random number generator used for domain generation appears to be a non-standard system, suggesting higher levels of software development experience. The author(s) also reuse a niche implementation of CRC32 [13] in their mutex name generation algorithm, a slightly modified variant of the original scheme. Such minor variations make it much harder to re-implement a routine. Furthermore, recent variants at least make intensive use of indirect calls and jumps and pick functions to pieces which significantly complicates analysis. The developers apparently knew their code would be reverse-engineered and prepared it in this way to make the analysts' job harder, which suggests an awareness of whitehat R&D. The author(s) also seem to know exactly which algorithm to use for which purpose. This requires staying current with the latest state-of-the-art in these technologies, which is another indication of their level of experience and professionalism.

Although this may appear to be praising malcode developers, their use of the hooking approach is technically very impressive. Conficker implements a generic routine that calculates the size required for arbitrary hooks (a JMP instruction in most cases), identifies all instructions that are modified using an integrated length disassembler, copies these instructions to a backup location and appends a jump back to the first unmodified instruction in the original code. This is a smart solution and definitely suggests strong domain knowledge, since there are very few legitimate use cases for this kind of approach in "normal" development.

Since version .B, Conficker has included virtual machine detection capabilities. It evaluates the result of the SLDT instruction to determine whether it runs in a virtual environment. While SLDT is very common in viruses, we have never seen it in *shellcode* before – Conficker.B's exploitation attempts contain it as well. Given this, it is all the more remarkable that the instruction's return value is not evaluated in the shellcode. So why was it included? Maybe checking the result was simply forgotten. Another possible answer is that

SLDT was used for *libemu* evasion, as discussed in section 1.2. This would be the first time we are aware of that exploit shellcode tries to evade this tool, indicating once again that the developers are up to date with current attack and malware analysis techniques. The authors also seem to track the news on Conficker and find solutions to evade countermeasures. One example of this progression is the much larger list of generated domains in Conficker.C, significantly increasing the cost and logistical effort required for system defenders to pre-register domain names (and only using a small fraction of those registered domains) . It is also interesting that they use query technologies like Maxmind's database and whois to find and evade their opponents' netblocks, and that they have been able to consistently automate mass registration of so many domain names.

Other technical features, such as the doubly-packed binaries, the thread injection capabilities and the fact that a memory image of Conficker does neither contain a valid PE header nor import table (to hamper dumping) are not that uncommon in modern malware, but still demonstrate that this piece of malware is state-of-the-art. Given that much of the malicious activity on the Internet today is aimed at financial gain, it at least seems likely that Conficker could have long term criminal goals.

14. CONCLUSION

In this paper, we have presented different ways to detect, contain and remove Conficker. The methods include local and remote detection, vaccination, as well as different approaches for identification, removal and prevention of local infections.

All Conficker variants try to patch the infected systems to prevent re-exploitation. The handler, installed as a function hook, changes the behavior of RPC requests on infected machines. This information can be used to remotely scan for Conficker infections. In addition to actively scanning, machines infected with Conficker.A and .B can be identified using the presented IDS signatures. Another option to remotely detect infected machines are the domain names generated by the different variants. By registering and sinkholing requests, infected machines can passively be enumerated. The domain name generation algorithm has been explained in detail including its custom PRNG. The domains cannot be registered for all IP addresses. Conficker.B and .C use blacklists to avoid AV and security companies and Microsoft networks, as discussed in Section 87. We have presented an approach that can terminate and wipe Conficker from memory while keeping the infected system services running. In addition, we have shown the file name generation algorithm that we extracted from Conficker. This information can be used to remove the infections. In order to prevent re-infection, as seen by several organizations, we have presented the *Nonficker Vaccination tool*, which prevents infections by using the mutex generation algorithm found inside Conficker. The final section of this paper allegorized a view of the Conficker authors' technical profile.

The original paper included a detailed explanation about issues in Conficker, which allow exploitation. The Conficker Working Group (CWG) has requested to not include this section in this public version for various reasons. The full paper will be published in the near future.

All descriptions include detailed information about the algorithms used. We have developed tools based on the extracted algorithms. The tools are released under the GPL and can be freely downloaded, together with their source code, from <http://iv.cs.uni-bonn.de/conficker> [9] .

ACKNOWLEDGEMENT

We would like to thank Stephan Schmitz and Christoph Fischer for valuable discussions about using mutexes to prevent Conficker infections. We are also grateful to all reviewers, especially to (in alphabetical order) Camilo Viecco, Christian Seifert, Dave Dittrich, David Watson, Georg Wicherski, Jose Nazario, Lance Spitzner, Markus Kötter, Mark Schloesser, and Paul Bächer of the HoneyNet Project, who provided extremely helpful comments, critical discussions and all the feedback. Markus and Paul also worked with us to add the SLDT instruction to *libemu*. The `Request` class in our scanner tool is based on code written by Georg. The memory scanning routine was taken from a previous joint work with Georg and Mark. Thank you. Furthermore, we would like to thank Dan Kaminsky for asking us if there is a way to detect Conficker remotely and later intensively testing the resulting network scanner prototype.

REFERENCES

- [1] Wikipedia: Blaster. <http://en.wikipedia.org/wiki/Blaster> (computer worm)
- [2] Wikipedia: Sasser. <http://en.wikipedia.org/wiki/Sasser> (computer worm)
- [3] Technet, M.: Microsoft security bulletin ms08-067.
<http://www.microsoft.com/technet/security/Bulletin/MS08-067.msp>
- [4] Porras, P., Saidi, H., Yegneswaran, V.: An analysis of conficker's logic and rendezvous protocol. Technical report, SRI International (2009)
- [5] Sotirov, A.: Decompiling the vulnerable function for ms08-067.
<http://www.phreedom.org/blog/2008/decompiling-ms08-067/>
- [6] uNderX: Micro length-disassembler engine 32. <http://vx.netlux.org/vx.php?id=em24>
- [7] Young, E.A., Hudson, T.J.: Openssl: The open source toolkit for ssl/tls. <http://www.openssl.org/>
- [8] Kleinjung, T., Franke, J. <http://www.crypto-world.com/announcements/m1039.txt> (2007)
- [9] F. Leder, T. Werner: Containing Conficker - webpage and tools. <http://iv.cs.uni-bonn.de/conficker>
- [10] Microsoft MSDN: Filetime structure.
[http://msdn.microsoft.com/en-us/library/ms724284\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms724284(VS.85).aspx)
- [11] Rekhter, Y., Moskowitz, B., Karrenberg, D., Groot, G.J.d., Lear, E.: RFC 1918: Address allocation for private internets (1996)
- [12] MaxMind: Geolocation and online fraud prevention. <http://www.maxmind.com>
- [13] Unknown: Possible origin Conficker.B and .C modified CRC32 algorithm,
http://read.pudn.com/downloads70/sourcecode/game/253142/bwh-1.5/source/bwh_setup/crc.cpp.htm
- [14]. Microsoft Corporation: Msdn – createmutex function.
[http://msdn.microsoft.com/en-us/library/ms682411\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682411(VS.85).aspx)
- [15] Stephen Lawler: reversing the ms08-067 patch..., <http://www.dontstuffbeansupyournose.com/?p=35>
- [16] libemu, <http://libemu.carnivore.it/>
- [17] "nebula – An Intrusion Signature Generator", <http://nebula.carnivore.it/>
- [18] snort, <http://www.snort.org/>

APPENDIX A

Conficker.A	Conficker.B	Conficker.C
CF F0 7B 61 89 D7 16 E8 1B 09 ED 31 45 FE 2E E7 8A 24 B9 56 F6 73 41 36 78 EF 37 50 DF EC 86 CA 4F E2 96 4B C3 E6 F4 50 BA 16 56 E8 4E 76 F2 5B 72 45 57 98 7A 07 0B 97 15 17 E9 AB F6 6D 13 56 DE 66 1A 55 AE AE 9E 94 53 EE 60 25 34 FC 01 CB F7 66 1D F4 B0 E9 1D 6F E9 A9 1B 82 C1 A3 5C 6E E3 DA 61 65 28 AB 36 6A A5 3E E9 EE 0A C1 3A E2 27 43 F6 1E 0B 03 2A 3C 9B 91 FE E9 40 76 BF 25	E7 A7 2D F5 45 CA 12 49 E6 44 1E D6 72 4C 1B BA 3C 72 F0 8B 4B EB 75 F3 5E E8 44 CD 87 56 E9 21 E6 06 34 33 76 49 93 42 EC E8 03 36 19 A6 AD 4D 12 59 7F 96 01 85 41 25 CB E2 83 7E 72 DF 85 B3 DD E1 59 FB 97 78 9A 2D B2 B6 3D E9 58 52 45 39 1B 90 C8 9F D7 5C 2B 42 DE A6 6A D8 03 D0 F2 4C AF 72 24 2E 9D 8C F3 4D 2F 2F 4D F2 49 D6 89 29 A2 C9 C6 FF F2 5F 98 B6 68 09 AD 92 10 70 D5 10 EA 1C DA B6 BC D4 03 CC 8D 9E 8E 57 8C CF FC BC 5B C3 9E 31 5B DA 08 8A 93 26 80 BF D2 DB 45 80 83 33 87 AF 69 C2 F6 5F 15 99 34 14 CB 0F 88 CC 44 29 E9 93 45 9E 7E F9 12 87 8A 93 8E 33 43 BB 0C 40 5B 60 4C 86 40 31 17 99 65 13 E4 6C C2 8A E5 A4 30 D9 F3 D6 6A BB EB DF DA 02 EC 6D 38 7E 23 EE 11 68 8A 62 7D A8 93 93 9E C6 DC 7B F1 23 5D 66 72 39 C8 3D E5 56 C3 20 D9 A8 9A 25 35 E4 3B 99 D4 7E 61 D1 D7 74 50 E3 6A EB 49 5A 31 3D 21 DE 29 4A CD 30 FC D1 FD D7 98 73 60 4B A6 53 08 5D F9 EE 05 E6 21 97 ED D9 B7 D6 BC 00 34 B1 76 6B BD 26 60 8A 2C 1C B6 E6 58 2D 47 4D 40 09 5B 83 B1 9D 3C 98 8E A2 2D 9E 5D 7A 07 F1 0D C8 1B 26 47 B0 1A 1C 70 08 76 4C C2 9C CF 3A F3 0E 1E C6 00 A8 15 CB 47 92 7E 1D F9 07 42 AA 92 49 DC 04 71 ED D6 E7 DC E6 AD 3C BD 25 18 32 FA EC FA B7 A5 FB CC A1 49 52 BA 30 60 A7 D3 B0 A3 95 E5 F2 DA 61 BD 27 D2 97 C0 D8 66 33 37 04 13 D2 36 9D 3F CB 24 79 6B 2E 69 22 E1 0B AD C1 5B 48 8A E1 D5 00 87 37 44 06 F5 AE 4C 74 4B 20 0F A3 57 63 08 D4 57 EB F0 3A E3 1A 03 C4 DF 9A 17 2D 7F FD 1F 44 71 DA 49 B7 BA 3F 26 B5 DD 9C FE CA 18 70 DB EC 99 63 84 96 30 10 80 4C 33 73 4D BC B2 C3 79 2C 83 68 CD 41 5C 45 ED 7D E7 BE A8 88	13 E5 A4 19 AD 58 46 33 10 1E FE C1 A1 96 4F B1 18 FB E1 49 20 32 9C 17 E8 1C 7C BC 6D 81 31 C9 DB 8D 7D BC A1 BB 94 E4 21 04 7D 30 AA F9 67 8C 78 A6 41 47 CA 08 57 B9 52 00 A9 2B E0 78 D7 FC 5A 57 FB 7C D9 D0 16 E2 2E 2C DE F6 DB 84 94 43 E7 EE 72 1C 86 7E 81 95 59 CE 39 BB A2 20 A0 81 63 03 09 3D 9C EF 5F EE 03 8B D9 56 21 A8 C4 FA B5 B1 69 20 74 30 B5 1C EA 83 04 78 50 78 AC 1C 8C 54 5D 2B E3 92 75 BD D9 78 5E EB 21 43 9B A6 16 89 D4 6B 0B 0F 0D 6A 15 47 F8 19 50 41 62 2C 9E D8 9A 56 3B 41 7A CA 5A 44 AF FB D6 7D F9 1C C0 46 A3 F5 81 49 BA D6 E1 06 A8 93 F6 E7 87 83 62 E8 C0 BC 2D 18 7A 69 1E 1A F5 19 5A B9 CE 7A 34 DC A0 3A 2E EE 6B 12 B2 F0 2D 2E 87 6D 5F 2D F5 97 BC F0 53 1A 30 C5 6D C5 A4 A2 28 BE 88 AD E1 79 06 6E 3C 90 F3 36 F9 2F F8 05 3B 5B EA C3 D3 01 E5 67 6C 4C 28 7E 35 B3 CD 6E 33 D3 D3 20 2E 09 CF 30 79 4A 41 03 62 76 1D 92 B2 A4 34 88 09 44 14 55 14 E8 EF F0 FC CB 82 E0 04 50 42 36 DC 3D B8 92 E9 AB 13 5A 7B 0E 07 A4 EB 71 69 1C 03 A9 0E 5E 29 43 57 D4 A8 E0 71 BD 47 6E 73 CC 82 0B 63 81 89 6F AC 5E 59 16 3F E2 D2 DC A4 BD F7 77 EA 0A F3 CC 6F 4D 51 DB 7C 94 7D 57 5E 1B A6 A2 18 E1 C5 B6 E2 20 8E 6C 11 4F C9 1E 76 56 93 8F 56 62 A0 11 86 1F 1D 1D 2D D7 2A E1 30 31 30 1C A8 BF 28 0D 0E 90 DA 2D 54 B7 8C 03 44 00 7D 5B AB C2 5F 51 34 E0 A5 BF 2D ED C3 28 D4 BB 59 B5 F2 5E FD 5A 5B 9A E9 39 62 B4 99 4E 35 09 E2 A2 B1 81 2B 2F 4B DB 3E 76 62 22 52 FD 79 5B 6E 53 70 3C BE 9E 4D AB 54 D3 9E 62 FF 59 7A 60 FC 78 48 1E 05 2F CB 1C F9 5F 23 91 FE 7E 42 E2 88 B0 D3 33 87 41 00 72 4F 94 63 22 67 B9 A2 20

Table 4: RSA Keys as stored in Conficker