Conficker.C
A Technical Analysis

Niall Fitzgibbon and Mike Wood
SophosLabs, Sophos Inc.

April 1, 2009

Abstract

The Conficker worm has grown to be one of the most technologically advanced and resilient botnets to date. While the initial worm variants, *Conficker.A*, *Conficker.B* and *Conficker.B++*, had a primary focus on spreading infection, the latest variant *Conficker.C* demonstrates a paradigm shift – moving away from overt infection tactics toward stealthly and robust operations. This paper will discuss the evolution of the Conficker malware family with special focus on the technological advancements in *Conficker.C* that have turned the millions of compromised machines from isolated infections to a collective of self-organizing peers capable of rapid malware distribution and resilient against infiltration of their communication paths.

DISCLAIMER

This is a living document containing the results of our analysis of the *Conficker.C* worm to date. As such, the details presented here remain subject to ongoing improvement and corrections.

Contents

1 Introduction

The Conficker family of malware is currently restricted to Windows machines. The key attack vector used by *Conficker.A* is the MS08-067 RPC NetPathCanonicalize vulnerability. This variant first appeared mid November 2008, followed by *Conficker.B* in late December 2008 and *Conficker.B++* in February 2009. The most recent variant, *Conficker.C*, was first seen at the start of March 2009 when it was installed as an update on computers already infected with *Conficker.B* and *Conficker.B++*.

Conficker.B and *Conficker.B++* were responsible for the major growth in the size of the Conficker botnet, as they were able to spread using Windows file sharing and `autorun.inf` files on USB media, in addition to the MS08-067 vulnerability used by *Conficker.A*.

The initial variants used clever social engineering for a secondary attack vector via autorun files in removable storage devices (e.g. USB drives). The worm also spread via network shares, pummeling the network with netbios activity.

Interestingly, the new *Conficker.C* has replaced the above spreading functionality with a more robust peer-to-peer (p2p) content distribution system. The peer-to-peer mechanism enables a *Conficker.C* infected host to share executable content between its peers. This essentially makes *Conficker.C* hosts capable of executing arbitrary content – a dramatic increase to the flexibility and potential for harm. In brief, the peer-to-peer mechanisms in *Conficker.C* allow an infected machine to:

- find other *Conficker.C* peers using a unique IP-to-port address mapping
- distribute or receive content, cryptographically signed only by the Conficker author(s)
- execute received content, for which the cryptographic signature is verified

Peer-to-peer functionality is discussed in greater depth in Section 3.

In the following sections we discuss several of the key advancements in the latest variant, *Conficker.C*, in the ongoing evolution of this malware family.

2 Armor

Conficker.C contains several features designed to prevent its removal from an infected computer. It also makes use of compiler-level obfuscation to make reverse engineering more difficult and contains code to detect whether it's running inside a virtual machine. Furthermore, *Conficker.C* hooks the API

function that is responsible for the MS08-067 vulnerability that previous variants of Conficker exploited in order to spread.

2.1 Removal prevention

In an effort to make removal of the worm more difficult *Conficker.C* attempts to

- lock the *Conficker.C* DLL file on disk to prevent it being scanned or deleted
- remove registry permissions from the service key for the *Conficker.C* DLL file to prevent it being examined or deleted by the Administrator account
- disable Windows services related to updates and security
- block access to security-related websites to prevent Windows and anti-virus software updating and to stop the user downloading removal tools
- terminate processes related to security diagnostic tools and Conficker removal tools

2.1.1 Locking the DLL file on disk

Conficker.C uses the `LockFile` Windows API to prevent other applications from reading or writing to the *Conficker.C* DLL file until the process holding the lock is terminated. This may cause complications when scanners attempt to read, detect and remove *Conficker.C*. The lock may need to be broken before detection and removal are possible. Generally, the *Conficker.C* DLL will be loaded inside `svchost.exe -k netsvcs`, a legitimate Windows service process, and terminating the process will break the lock on the file.

2.1.2 Service registry key permissions

After creating the service registry entries to load the *Conficker.C* module as part of the `netsvcs` service group, the worm will manipulate the access control list (ACL) of the service keys, restricting the `SYSTEM` user to read access and removing permissions for every other user. This prevents all users, including local and domain administrators, from opening or deleting the registry key.

2.1.3 Disabling Windows services

Conficker.C disables the following services:

- `wscsvc` - Security Centre
- `WinDefend` - Windows Defender
- `wuauserv` - Automatic Updates
- `BITS` - Background Intelligent Transfer Service
- `ERSvc` - Error Reporting Service
- `WerSvc` - Windows Error Reporting

The worm removes the following registry key to prevent Windows Defender from running on startup:
`HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Windows Defender`

A registry key is removed to suppress Windows Security Centre notifications:
`HKLM\Software\Microsoft\Windows\CurrentVersion\explorer\ShellServiceObjects\{FD6905CE-952F-41F1-9A6F-135D9C6622CC}`

Conficker.C deletes the following key to disable booting Windows in safe mode:
`HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot`

2.1.4 Blocking websites

Once loaded *Conficker.C* hooks the following Windows API functions from `dnsapi.dll` in order to intercept DNS requests: `DnsQuery_A`, `DnsQuery_UTF8`, `DnsQuery_W` and `Query_Main`. When a DNS request occurs, *Conficker.C* checks for matches against a list of blocked domains. If there is a match, *Conficker.C* calls `SetLastError` with a value of `ERROR_TIMEOUT` and then returns this error without calling the original function.

Conficker.C also hooks the `ws2_32.dll` function `sendto` as long as the `dnsrslvr.dll` module is loaded in the current process. The `sendto` hook function checks if it is being called from `dnsrslvr.dll`. If the call comes from `dnsrslvr.dll` then *Conficker.C* will check if the domain name is in the blocked domain list and if so will replace it with a randomized string.

2.1.5 Terminating security processes

Conficker.C continuously monitors the list of running processes. For each process, the worm checks for a substring match on any of the strings in its table of blacklisted process names. For processes with names in the blacklist, *Conficker.C* will first suspend all threads belonging to the process and then call `TerminateProcess` to kill the process itself.

2.2 Obfuscation

Like much recent malware, *Conficker.C* uses obfuscation in several places. Firstly, the DLL file itself is encrypted and will decrypt itself into newly allocated memory within the current process when it starts up. The decryption routine is polymorphic and makes heavy use of redundant API calls and "spaghetti code", where functions are split up and connected by x86 `jmp` instructions (both direct and indirect). The spaghetti code obfuscation makes static analysis more difficult as it is hard for the analyst to view a linear disassembly listing for each function, and they must resort to more advanced disassembler features such as graph views. Redundant API calls are somewhat effective at preventing accurate emulation by anti-virus engines, as it is unlikely that they have a full and correct implementation of all possible Windows API functions and while they may properly emulate the loading and unloading of stack arguments and return values for most functions, they are less likely to properly take account of secondary effects such as error values (available from the `GetLastError` Windows API).

Once decrypted, *Conficker.C* differs from previous variants in that a portion of its unpacked code is also obfuscated with the same spaghetti code technique. However, the unpacked code does not contain fake API calls and can be analysed relatively easily once the spaghetti code is accounted for. APIs used in the obfuscated code are dynamically imported, with the resolved API addresses being stored in dynamically allocated memory outside of the main *Conficker.C* module. The code protected by this newly introduced obfuscation is mainly related to the peer-to-peer networking functionality.

2.3 Virtual machine detection

Conficker.C contains several different checks to determine if it is running inside a virtual machine environment. These checks are split over two areas of the *Conficker.C* code.

The first check takes place at the start of the main initialization thread, before most other initialization has taken place. *Conficker.C* executes the `sldt` instruction to store the segment of the local descriptor table in a register. If the result is a non-zero value, *Conficker.C* assumes that it is running in a virtual machine and executes a `Sleep` API call with time set to `INFINITE`.

A later set of checks occurs during initialization of the peer-to-peer networking functionality. If any of these tests suggests that the malware is running inside a virtual machine, then *Conficker.C* will not terminate but instead will set corresponding bitfields inside a block of platform information that is later transmitted over the peer-to-peer network. The further checks done in this function include

- a repeat of the initial `sldt` check mentioned above

- a Red Pill-style check using `sidt` that is continuously repeated for one second, only passing if the value returned from `sidt` didn't indicate the presence of a virtual machine once in that time
- a test using the `str` instruction, comparing the stored task register with `4000h`
- a test using the `sgdt` instruction, checking if the result is in the range `0ff000000h` to `0ffffffffh`
- a test using the `in` instruction with parameter `'VMXh'` to check for the presence of a VMWare virtual machine
- a test using the illegal instruction `0f 3f 07 0b` which would normally trigger an illegal opcode exception, but which is known to be handled differently if the program is running inside VirtualPC

2.4 Patching MS08-067

Once loaded, *Conficker.C* hooks the `NetpwCanonicalizePath` export of `netapi32.dll` that is central to the exploitation of the MS08-067 vulnerability. The hook function contains a validation check that effectively patches the computer against further exploitation via MS08-067 as long as *Conficker.C* is loaded. If the argument passed to `NetpwCanonicalizePath` contains the substring "\..\" or is over 200 bytes in length, the hook function will call `SetLastError` with error value `ERROR_INVALID_PARAMETER` and return this error value without calling the original `NetpwCanonicalizePath` function. Other security analysts have developed a remote scanning system that identifies computers infected with Conficker by analyzing the effects of these extra failure conditions on network responses from an infected computer.

3 Peer-to-Peer Activity

Conficker.C includes a p2p mechanism, enabling *Conficker.C* infected hosts to distribute signed Conficker-only content between peers. This content can also be executed as a new thread in the same address space as the *Conficker.C* process. This section discusses the various aspects of the p2p mechanism we have observed to date, including the initial setup, bootstrapping peer connections, and file sharing

3.1 Setup

The p2p setup is called from the main Conficker installation thread after all of the main installation actions, such as

- hooking certain Windows APIs
- installing itself in the registry
- injecting itself into the svchost or services process
- locking the file on disk
- disabling security tools

However the p2p system is started before the callhome protocol. After the above steps, *Conficker.C* sleeps for a random 5 to 35 minutes before attempting to start p2p networking, and further sleeps for random 30 to 90 minutes after p2p networking has begun before starting the callhome procedure.

The p2p process begins by saving the time and tick count when the process begins. This value is used extensively throughout the p2p protocol, but particularly when determining when the lifetime of a file managed by the node has expired (and should be deleted).

Next, the directory used to store p2p managed content is created. The `GetTempPath` function is used to retrieve a base directory. As described in the MSDN documentation, this function will enumerate the environment variables `TMP`, `TEMP` and `USERPROFILE` in that order looking for a directory name before defaulting to the system `WINDOWS` directory. A sub-directory of this base directory is then created with a name using the following format string:

```
{%08X-%04X-%04X-%04X-%08X%04X}
```

For example, the p2p directory name observed on one of our test systems was

C:\WINDOWS\Temp\{27623480-9BE0-244C-E6CD-E64B466BBDE2}

3.2 Peer Serving Logic

Conficker.C can act as both a client and a server to share content, both distributing to and receiving from other *Conficker.C* peers. An instance of a *Conficker.C* p2p server consists of the following threads:

1. a time synchronization thread – used to set the system time using a time value parsed from an HTTP GET request made to one of a long list of legitimate domains;
   the requests are repeated in a loop, separated by a random interval of 90 to 120 seconds
2. four scanning threads – 2 TCP, 2 UDP – used to actively locate other *Conficker.C* peers

3. a main server thread – used to passively listen for scanning *Conficker.C* peers

The main server thread creates 4 sockets – 2 TCP and 2 UDP – to listen for incoming connections from scanning *Conficker.C* peers. These sockets are bound to port numbers that are derived from the IP address on which the sockets listen. The algorithm to generate these port numbers creates four unique ports on which to listen – two for TCP sockets and two for UDP sockets.

The four scanning threads generate a random list of IP addresses to contact. The same IP-to-port mapping is used to determine which ports on which IPs should be targeted. The behavior of these threads and the IP-to-port algorithm is further discussed in Section 3.3.

A *Conficker.C* infected host can have as many as 32 confick server instances running. One server instance will be created for each IP address that is not filtered out by *Conficker.C*'s IP blacklisting techniques.

3.3 Finding peers

Four scanning threads form the active scanning component of *Conficker.C*'s p2p mechanism. For each scanning thread, 100 IP addresses are generated and probed in a batch with the probes in each batch separated by small fixed intervals of 2-5 seconds.

Unlike many p2p networks that require an initial seed list of peers, *Conficker.C* is designed to bootstrap communication with other *Conficker.C* peers using a deterministic IP-to-port address mapping routine. For each generated IP address to be probed, the current internet time (as managed by the internet time thread) is used as a seed value to mix the bytes of the IP address to deterministically compute four port numbers – two for use by UDP sockets, and two for TCP sockets.

Each time an IP-to-port mapping is calculated, the internet time value is checked to determine whether to re-seed the algorithm. The seed is re-calculated if 60 seconds has elapsed since the last time the seed was set. However, the computation will only change the seed value every 604801 seconds – which is 7 days worth of seconds plus 1. Thus, *Conficker.C* server port numbers change on a weekly basis.

Naturally, this IP-to-port mapping is the same computation used in the server main thread where the four listening sockets are created. The main thread also checks periodically for changes to the port generation seed, closing and re-opening the listening sockets when a change is detected. During a *Conficker.C* host scan, one of the two ports generated for the protocol in use (TCP or UDP) is randomly selected for the probe. Though, this random selection does not impact the success rate of finding peers since a *Conficker.C* server (if one exists at the IP being probed) will be listening on both ports generated.

Interestingly, when computing this weekly dynamic seed, the current time must fall between Jan 14, 2009 and Jan 15, 2015. If the current time falls outside this range, fixed seeds are used for the port number calculation. Thus, using this date range as an indicator for the Conficker author's intentions, he, she or they appear(s) to believe this protocol will stand the test of time for several years to come.

3.4 Shared Content

All content shared between *Conficker.C* peers is checked for a valid digital signature. The signature validation process is the same as that used in the Rendezvous Protocol (see Section 4), however a separate RSA key is used for p2p binary validation.

Shared content between *Conficker.C* peers is stored as files in the p2p directory described in Section 3.1. This directory is restricted to hold a maximum of 64 files. If this maximum is reached, the directory is cleaned of any files not created within the last 10 minutes, or any files with a write time not within the last 1 minute. The file names are numbers, representing the location in the list of *Conficker.C* shared binaries, and will have the form "%d.tmp" (sprintf format).

Shared content can also be executed as a new thread within the *Conficker.C*'s current process address space. This ability obviates the need for the reinfection backdoor mechanism seen in previous variants.

4 Domain Rendezvous ("Callhome") Mechanism

Previous variants *Conficker.B* and *Conficker.B++* incorporated a self-updating mechanism via pseudo-randomly generated domain names. In short, Conficker includes an algorithm to deterministically generate a set of domain names each day, and contact those domains throughout the day to check if an updated version of the worm was available for download.

4.1 Protocol

In *Conficker.C*, this rendezvous protocol was updated as well, to generate a massive 50,000 potential call-home domains per day (over and above the 250 per day seen in the previous variants). Although *Conficker.C* generates 50,000 domains per day, the worm randomly chooses only 500 of those domains to rendezvous with that day. Furthermore, it only attempts to resolve each of those 500 domains once during the day. This is in stark contrast to the previous *Conficker.B* scheme, which generated 250 potential call-home domains and repeatedly queried all 250 domains once every 2 hours. Thus, *Conficker.C*'s use of its callhome domains results in far fewer DNS queries than that of prior variants (only 500 per day vs. 3,000).

	Variant B	*Variant C*
Domains / day	250	50,000
Used / day	250	500
Query interval	Every 5 seconds	Random $[10, 50]$ second interval
Process repeats	Every 2 hours	Once per day
Total DNS queries / day	3,000	500
Enabled on	Jan 1, 2009	Apr 1, 2009

Table 1: Comparing rendezvous protocols: *Conficker.B* vs. *Conficker.C*

The DNS query intervals are also randomized in *Conficker.C* to make the rendezvous protocol more stealthy. Prior variants issued a small number of DNS queries in parallel at a fixed 5 second intervals. *Conficker.C* however issues DNS queries in series and separates each query with a random interval between 10 and 50 seconds. This is likely an effort to avoid triggering anomaly detection systems that can pick up spikes or fixed patterns in network activity. For a detailed comparison between the rendezvous protocols in *Conficker.B* and *Conficker.C*, see Table 4.1.

4.2 Blacklisting

Previous variants blindly attempted to download a conficker payload from a callhome domain after resolving the domain to an IP address. However, *Conficker.C* filters each IP address of resolved callhome domains before attempting to download an update. The IP address will be rejected if:

- it is a private IP address (192.168.*, 10.*, 172.16.*)
- it is a reserved IP address or multicast address
- it matches an internal IP blacklist

4.3 What happens on a successful rendezvous?

For a Conficker infected host (of any variant), a successful rendezvous is comprised of the following sequence of events, where the infected host:

1. retrieves a binary data file from a callhome domain for the current day, using the IP address to which the callhome domain resolves to fetch the url directly

   ***Conficker.B* URL** http://*ip-address*/search?q=N
   where N is an integer

   ***Conficker.C* URL** http://*ip-address*/

2. verifies the digital signature on the encrypted payload as that of the Conficker author via the rendezvous protocol RSA public key (public keys listed in Section 8.1)

3. decrypts the encrypted payload and verifies the hash on the decrypted data, as decrypted from the RSA signature

Upon successfully completing all of the above steps, the Conficker host writes the decrypted content to a file and executes the file using `CreateProcessA`. Furthermore, after a successful rendezvous, the infected *Conficker.C* host sleeps for an additional 3 days before resuming the above callhome activity.

5 Cryptography

The Conficker malware family demonstrates effective use of modern cryptography to make injecting innoculating commands into the network of infected machines far more challenging. The signature mechanism appears generally robust to attack, using standard RSA for signature validation with 4096 bit modulii (very difficult to factor) and high public exponents (to thwart other low exponent attacks). Strong randomization is also used throughout – employing `CryptGenRandom` using the "Microsoft Base Cryptographic Provider v1.0" or time-seeded `srand,rand` sequences when said cryptographic provider is not available.

Conficker.C is no different in this respect from previous variants, using the same signature validation and hashing process as *Conficker.B* for binaries downloaded from rendezvous domains. However, *Conficker.C* uses this signature validation to a greater extent than prior variants, including it in the p2p distribution of executable content as well as for registry values created by the worm. The signed registry values undergo an additional layer of custom obfuscation prior to being stored and the reverse de-obfuscation prior to being validated using the p2p RSA public key (see Section 8.1 for cryptographic key details).

6 Discussion

While Conficker has successfully infected millions of machines, what exactly this massive network will be used for is yet unknown. From our analysis, and from the research of other members of the security community, the latest variant *Conficker.C* is a robust platform for rapid p2p distribution of binary content.

The nature of the shared content between *Conficker.C* peers is not yet known. However, the treatment of the shared content allows us to make inferences as to its purpose. For instance, the p2p mechanism optionally employs `CreateThread` to execute shared content within the same address

space as *Conficker.C* – suggesting the shared data will merely augment the overall functionality of *Conficker.C* rather than replace it entirely. Further to this point, files stored in the p2p directory have a short lifespan; files are deleted if they were created more than 10 minutes prior, if space is needed for new shared files; and the maximum number of files stored is a relatively small 64. This design targets a high turn-over rate of shared content and suggests the p2p mechanisms may be used to distribute malicious commands similar in nature to how a botnet operates.

In contrast, the rendezvous protocol is likely to continue to be used as a "major update" channel by which the Conficker platform is updated, in a similar manner as seen before with the move from *Conficker.B* and *Conficker.B++* to *Conficker.C*. This seems logical, as from the analysis above, a successful rendezvous results in an entirely new process on the host with a call to `CreateProcessA` followed by a prolonged delay in callhome activity.

7 Conclusion

This report details several of the technical advancements in the lastest Conficker variant. Our report is one of many efforts from the security community. We hope our results will help the general public and other members of the security community to better protect against this worm. However, although our understanding of this malware family continues to improve, with the massive number of world-wide infections and the ability for peers to efficiently distribute arbitrary payloads amongst one another, it is likely that Conficker malware will continue to remain a genuine threat.

8 Appendix

8.1 RSA Public Keys

***Conficker.B* rendezvous public key**

```
Modulus (4096 bit):
    00:88:a8:be:e7:7d:ed:45:5c:41:cd:68:83:2c:79:
    c3:b2:bc:4d:73:33:4c:80:10:30:96:84:63:99:ec:
    db:70:18:ca:fe:9c:dd:b5:26:3f:ba:b7:49:da:71:
    44:1f:fd:7f:2d:17:9a:df:c4:03:1a:e3:3a:f0:eb:
    57:d4:08:63:57:a3:0f:20:4b:74:4c:ae:f5:06:44:
    37:87:00:d5:e1:8a:48:5b:c1:ad:0b:e1:22:69:2e:
    6b:79:24:cb:3f:9d:36:d2:13:04:37:33:66:d8:c0:
    97:d2:27:bd:61:da:f2:e5:95:a3:b0:d3:a7:60:30:
    ba:52:49:a1:cc:fb:a5:b7:fa:ec:fa:32:18:25:bd:
    3c:ad:e6:dc:e7:d6:ed:71:04:dc:49:92:aa:42:07:
    f9:1d:7e:92:47:cb:15:a8:00:c6:1e:0e:f3:3a:cf:
    9c:c2:4c:76:08:70:1c:1a:b0:47:26:1b:c8:0d:f1:
    07:7a:5d:9e:2d:a2:8e:98:3c:9d:b1:83:5b:09:40:
    4d:47:2d:58:e6:b6:1c:2c:8a:60:26:bd:6b:76:b1:
    34:00:bc:d6:b7:d9:ed:97:21:e6:05:ee:f9:5d:08:
    53:a6:4b:60:73:98:d7:fd:d1:fc:30:cd:4a:29:de:
    21:3d:31:5a:49:eb:6a:e3:50:74:d7:d1:61:7e:d4:
    99:3b:e4:35:25:9a:a8:d9:20:c3:56:e5:3d:c8:39:
    72:66:5d:23:f1:7b:dc:c6:9e:93:93:a8:7d:62:8a:
    68:11:ee:23:7e:38:6d:ec:02:da:df:eb:bb:6a:d6:
    f3:d9:30:a4:e5:8a:c2:6c:e4:13:65:99:17:31:40:
    86:4c:60:5b:40:0c:bb:43:33:8e:93:8a:87:12:f9:
    7e:9e:45:93:e9:29:44:cc:88:0f:cb:14:34:99:15:
    5f:f6:c2:69:af:87:33:83:80:45:db:d2:bf:80:26:
    93:8a:08:da:5b:31:9e:c3:5b:bc:fc:cf:8c:57:8e:
    9e:8d:cc:03:d4:bc:b6:da:1c:ea:10:d5:70:10:92:
    ad:09:68:b6:98:5f:f2:ff:c6:c9:a2:29:89:d6:49:
    f2:4d:2f:2f:4d:f3:8c:9d:2e:24:72:af:4c:f2:d0:
    03:d8:6a:a6:de:42:2b:5c:d7:9f:c8:90:1b:39:45:
    52:58:e9:3d:b6:b2:2d:9a:78:97:fb:59:e1:dd:b3:
    85:df:72:7e:83:e2:cb:25:41:85:01:96:7f:59:12:
    4d:ad:a6:19:36:03:e8:ec:42:93:49:76:33:34:06:
    e6:21:e9:56:87:cd:44:e8:5e:f3:75:eb:4b:8b:f0:
    72:3c:ba:1b:4c:72:d6:1e:44:e6:49:12:ca:45:f5:
    2d:a7:e7
Exponent: 50001 (0xc351)
```

Conficker.C rendezvous public key

```
Modulus (4094 bit):
    20:a2:b9:67:22:63:94:4f:72:00:41:87:33:d3:b0:
    88:e2:42:7e:fe:91:23:5f:f9:1c:cb:2f:05:1e:48:
    78:fc:60:7a:59:ff:62:9e:d3:54:ab:4d:9e:be:3c:
    70:53:6e:5b:79:fd:52:22:62:76:3e:db:4b:2f:2b:
    81:b1:a2:e2:09:35:4e:99:b4:62:39:e9:9a:5b:5a:
    fd:5e:f2:b5:59:bb:d4:28:c3:ed:2d:bf:a5:e0:34:
    51:5f:c2:ab:5b:7d:00:44:03:8c:b7:54:2d:da:90:
    0e:0d:28:bf:a8:1c:30:31:30:e1:2a:d7:2d:1d:1d:
    1f:86:11:a0:62:56:8f:93:56:f6:1e:c9:4f:11:6c:
    8e:20:e2:b6:c5:e1:18:a2:a6:1b:5e:57:7d:94:7c:
    db:51:4d:6f:cc:f3:0a:ea:77:f7:bd:a4:dc:d2:e2:
    3f:16:59:5e:ac:6f:89:81:63:0b:82:cc:73:6e:47:
    bd:71:e0:a8:d4:57:43:29:5e:0e:a9:03:1c:69:71:
    eb:a4:07:0e:7b:5a:13:ab:e9:92:b8:3d:dc:36:42:
    50:04:e0:82:cb:fc:f0:ef:e8:14:55:14:44:09:88:
    34:a4:b2:92:1d:76:62:03:41:4a:79:30:cf:09:2e:
    20:d3:d3:33:6e:cd:b3:35:7e:28:4c:6c:67:e5:01:
    d3:c3:ea:5b:3b:05:f8:2f:f9:36:f3:90:3c:6e:06:
    79:e1:ad:88:be:28:a2:a4:c5:6d:c5:30:1a:53:f0:
    bc:97:f5:2d:5f:6d:87:2e:2d:f0:b2:12:6b:ee:2e:
    3a:a0:dc:34:7a:ce:b9:5a:19:f5:1a:1e:69:7a:18:
    2d:bc:c0:e8:62:83:87:e7:f6:93:a8:06:e1:d6:ba:
    49:81:f5:a3:46:c0:1c:f9:7d:d6:fb:af:44:5a:c4:
    7a:41:3b:56:9a:d8:9e:2c:62:41:50:19:f8:47:15:
    6a:0d:0f:0b:6b:d4:89:16:a6:9b:43:21:eb:5e:78:
    d9:bd:75:92:e3:2b:5d:54:8c:1c:ac:78:50:78:04:
    83:ea:1c:b5:30:74:20:69:b1:b5:fa:c4:a8:21:56:
    d9:8b:03:ee:5f:ef:9c:3d:09:03:63:81:a0:20:a2:
    bb:39:ce:59:95:81:7e:86:1c:72:ee:e7:43:94:84:
    db:f6:de:2c:2e:e2:16:d0:d9:7c:fb:57:5a:fc:d7:
    78:e0:2b:a9:00:52:b9:57:08:ca:47:41:a6:78:8c:
    67:f9:aa:30:7d:04:21:e4:94:bb:a1:bc:7d:8d:db:
    c9:31:81:6d:bc:7c:1c:e8:17:9c:32:20:49:e1:fb:
    18:b1:4f:96:a1:c1:fe:1e:10:33:46:58:ad:19:a4:
    e5:13
Exponent: 50003 (0xc353)
```

Conficker.C p2p public key

```
Modulus (4095 bit):
```

```
    7a:73:1f:5e:c1:ae:c6:6b:b5:45:6c:d2:fc:b9:3e:
    c1:0a:39:d8:98:40:a1:41:19:e8:c5:ca:09:a9:c9:
    24:ea:94:41:5a:10:d1:03:a1:42:b3:a1:81:ca:02:
    17:54:02:ca:cc:d8:f6:26:7c:86:cb:af:a6:f8:97:
    61:28:70:e6:b0:b5:b5:cb:4f:c9:30:8e:4c:5e:d1:
    dc:74:f1:91:f6:01:e9:94:d6:30:8c:82:7f:77:51:
    8f:40:e3:41:81:93:ba:a4:a7:92:5b:09:5c:d6:d7:
    48:99:5e:c5:73:27:ba:65:ed:de:a3:8a:74:63:97:
    60:d9:16:86:86:c5:eb:d0:22:90:34:96:20:fa:ef:
    ab:fe:05:7a:5d:30:f0:bd:f5:c9:26:82:59:2b:e4:
    b7:63:0e:d0:77:7e:6e:d6:99:30:4b:30:54:15:0e:
    52:fa:e2:29:40:bc:51:02:42:79:8e:2d:a2:fa:13:
    56:47:60:17:b4:df:ae:8a:89:5e:e0:17:07:a9:6b:
    49:4e:78:ba:07:80:ed:bb:b5:c8:25:52:19:2c:95:
    35:45:c1:0f:77:63:39:af:3e:fc:ac:ba:c7:97:11:
    de:76:17:d3:c0:58:0a:20:69:07:9a:64:d3:b9:81:
    29:6d:5e:72:f5:24:03:5c:63:7b:c6:00:1c:81:eb:
    c5:94:13:cc:cf:d6:39:10:ce:62:4c:3a:23:fd:0c:
    8b:9e:f5:b9:f8:e0:60:fc:46:cb:ba:5a:27:6c:11:
    d7:33:a9:cc:7d:7d:89:49:35:76:51:4c:91:5d:bb:
    6e:74:ff:9b:7e:4d:28:f4:d7:d3:24:5d:1c:bb:fb:
    42:e9:aa:06:09:be:ea:ed:2b:89:5b:83:3b:20:ea:
    09:6a:4b:8d:7e:57:6d:54:84:2e:ed:1b:d1:25:13:
    4b:ed:c7:20:3e:ba:4e:74:0a:88:fb:e1:df:13:c1:
    a6:9e:3f:cf:5b:74:d3:e9:40:78:98:05:09:89:71:
    e6:73:e8:69:e6:70:53:b5:ea:0e:60:a6:93:7b:18:
    2c:3c:34:5c:1e:75:6c:c2:96:cb:32:0e:ff:67:c4:
    c1:af:27:b9:6f:11:a8:17:ca:0f:13:7b:ea:69:ac:
    ac:e6:f7:92:67:bf:65:8d:ee:9b:3c:dd:e9:46:eb:
    8c:0d:33:03:5c:61:c5:c2:93:76:83:f8:e5:02:f2:
    67:fa:a6:f6:a6:cc:a6:3e:06:1d:47:af:4f:e9:b7:
    82:ef:fb:68:f7:05:06:8e:d6:cb:7b:c6:49:e3:99:
    32:3f:20:6d:ad:7c:d7:78:eb:3c:d7:c5:04:8c:4f:
    d1:09:e1:5d:92:f5:4a:bf:9a:c9:cf:b1:e0:28:3c:
    c4:91
Exponent: 50005 (0xc355)
```