Tyler Cingel

Reverse Engineering

April 25th, 2022

Reverse Engineering of the Morris Worm

Method of Analysis

For external analysis, I will be using several websites. In general, these websites will include security databases, as well as articles and/or papers. I will be using the security databases to find technical details about the virus such as method of insertion into a machine, exploits involved in its execution, and how it operates. I will be using articles and/or papers for any non-technical details related to this virus such as when and how it was discovered, initial reactions to the virus, and effect it had on the world. A list of sources will be included at the end of this report.

For all self-analysis, I will be using an Ubuntu 64-bit virtual machine. I will be downloading the source code from this GitHub repository: https://github.com/arialdomartini/morris-worm, and will be examining the files in Notepad. Miscellaneous notes during analysis will be taken on Notability on an iPad, and from there the information will be condensed into what is relevant for the report. Creating signatures for the malware will be done using YARA, which will be installed on the Ubuntu virtual machine. A disclaimer about the documentation present in the source code of this file: I reached out to the uploader of this source code to inquire about who wrote which portions of the documentation, be it himself, Robert Morris, or some other researcher, but have received no reply. I believe the documentation is still valuable and credible and will be using and referencing it during my analysis.

The process of my analysis will start with me doing most of my external research about this virus. This will include learning about its background and how it functioned in a high level. The purpose of starting with the background is twofold. First, I want to understand the significance of this worm before I start my analysis in order to get a better appreciation of what I am looking at. Second, understanding how this worm works at a high level will help me navigate the code a lot easier, as I will know what to look for in certain respects and I will not be blindly reading code to decipher meaning. Following that, I will examine the code primarily through static code analysis. This will include looking over each of the individual files and functions and attempting to assign a purpose to each. Outside research will still be involved in the form of looking up known function calls that are from external libraries or are built-in to C. Finally, the final step of research will be developing a YARA signature suite to identify individual files, as well as the malware as a whole. Signatures will be created with a combination of critical and unique functions of the code (i.e., parts of the code where the program either cannot function without it, or that are unique to the Morris Worm).

Background

The Morris Worm was released into the wild on November 2nd, 1988, by a Cornell graduate student named Robert Tappan Morris. The worm was initially released onto a Massachusetts

Institute of Technology computer and from there is spread to over 6,000 computers connected to the internet. Since it was only designed for Unix systems (predecessor of Linux) 6,000 computers was its peak, however at the time this was still roughly 10% of all computers connected to the internet. According to Robert Morris the worm was originally intended to be used to gauge the size of the internet and was accidentally released with bugs that caused to have malicious side effects on the devices it infected. While it did not delete, encrypt, steal, or spy on any information on the computer, it did cause a Denial-of-Service effect (where a service becomes unusable for any given reason) due to the large number of processes and calculations the program performed on the system it infected. Despite Morris' claims that the release of this virus was accidental, he was still charged with violating the 1986 Computer Fraud and Abuse Act. He was not charged with any prison time, but still received three years of probation, 400 hours of community service, and a $10k fine for this incident.

The significance of this outbreak is due to a number of factors. For starters, Robert Tappan Morris was the first person to be charged with violation of CFAA Act as it had been created only 2 years before this incident. In addition to this factor, the Morris Worm was also one of the largest and most costly worms of its time and could even be considered one of the first worms. The cost of damages is hard to put a precise number on, but estimates claim it could have been anywhere from $100,000 to $1 million. Due to its large reach, it affected computers in several prestigious universities including John Hopkins, Stanford, Princeton, and Harvard, as well as affecting some computers at NASA. Another side effect, and contributor to its legacy, is the fact that this was one of the most covered hacking incidents of its time, garnering far more attention than any worm or virus that had come before it.

Intrusion Method

The first vulnerability this worm tries to exploit for intrusion is one in the remote execution of Unix mailing systems. It utilizes information from the user's ".forward" or "/.rhosts" files.

Figure 3.1 – "attack_user" code snippet

```
sprintf(buf, XS("%.200s/.rhosts"), user->homedir);
fwd_fp = fopen(buf, XS("r"));
if (fwd_fp) {                                   /* 2446 */
    while (fgets(rhbuf, sizeof(rhbuf), fwd_fp)) { /* 2312,2434 */
        l1288[0] = '\0';
        if (sscanf(rhbuf, XS("%s%s"), buf, l1288) < 1)
            continue;
        host = h_name2host(buf, 1);
        hu1(user, host, l1288);
    }
    fclose(fwd_fp);
}
```

In Figure 3.1 is a portion of the function "attack_user" in the file "cracksome.c" where the worm is passing along information gathered from the "/.rhosts" file to a function "hu1" (heuristic 1) in the file "hs.c". What is happening here is the contents of the file "/.rhosts", the name of which is in the variable "buf", is being transferred to the variable "l1288", and the valid hosts are being

gathered from the function “h_name2host” (which comes from the file stubs.c) are being sent to “hu1”. This function then attempts to exploit a remote execution vulnerability on Unix’s “sendmail” debug program.

Another method of intrusion is a stack-smashing attack using the “fingerd” daemon in Unix systems. The details of which are explained in the documentation left in the script.

Figure 3.2 – Fingerd exploit description

```
695 /* This routine exploits a fixed 512 byte input buffer in a VAX running
696  * the BSD 4.3 fingerd binary.  It send 536 bytes (plus a newline) to
697  * overwrite six extra words in the stack frame, including the return
698  * PC, to point into the middle of the string sent over.  The instructions
699  * in the string do the direct system call version of execve("/bin/sh"). */
700
701 static try_finger(host, fd1, fd2)                    /* 0x49ec,<just_return+378 */
```

The worm attempts this vulnerability using various configurations when calling the function “try_rsh_and_mail”, which itself is called in three different functions.

The final potential method of entry is an attack on weak passwords. For this alone this worm has three different tactics. This first tactic is to check the passwords “username”, “usernameusername”, and “emanresu” (username backwards) as the passwords for the system. Then, if none of those passwords are valid, it attempts to login using a hard-coded password list of common words used as passwords. If neither of those lists worked, then the final list is an external .txt file containing all words in the English dictionary.

Figure 3.3 – “try_words” iterating over the hard-coded password list.

```
390     if (nextw == 0) {
391         for (i = 0; wds[i]; i++)
392             ;
393         permute(wds, i, sizeof(wds[0]));
394     }
395
396     for (j = 0; wds[nextw][j] != '\0'; j++)
397         wds[nextw][j] &= 0x7f;
398     for (user = x27f28; user; user = user->next)
399         try_passwd(user, wds[nextw]);
400     for (j = 0; wds[nextw][j]; j++)
401         wds[nextw][j] |= 0x80;
```

Figure 3.4 – crackme.c opening and testing the dictionary file

```
/* Called only from the cracksome() dispatch loop. Tries a single word from th
e
 * dictionary, downcasing if capitalized and trying again. */
static dict_words()                                    /* 0x67f0 */
{
    char buf[512];
    struct usr *user;
    static FILE *x27f30;

    if (x27f30 != NULL) {
        x27f30 = fopen(XS("/usr/dict/words"), XS("r"));
        if (x27f30 == NULL)
            return;
    }
```

Actions

Not much can be said about what this worm does while on a system, other than it continues attempting to spread to other systems. The main loop of this virus that is always running is called "mainloop" and is in the "worm.c" file. It is a while loop with the condition being True, so the exit condition is handled elsewhere. The contents of this loop are first, calling the "cracksome" function in the cracksome.c file, which attempts to connect and break into other devices by finding or guessing passwords. After this, it changes its process id by forking and deleting the original process, likely a method of hiding itself. Next, it tries to connect and break into other devices using the other methods of intrusion mentioned earlier. Finally, it cleans the host list, and checks if a parameter has been set so that it can exit the loop.

Figure 4.1 – "mainloop" in worm.c

```
while (1) {
    /* Crack some passwords */
    cracksome();
    /* Change my process id */
    if (fork() > 0)
        exit(0);
    if (hg() == 0 && hi() == 0 && ha() == 0)
        hl();
    other_sleep(120);
    time(&time1);
    if (time1 - time0 >= 60*60*12)
        h_clean();
    if (pleasequit && nextw > 0)
        exit(0);
}
```

From the image above, the functions "hg", "hi", "ha", and "hl" are all intermediate functions that eventually call all the functions necessary to test their exploits, including "try_rsh_and_mail" and "try_finger".

So, if the worm does not do much while on a system, then how was a quarter of the internet reduced to a crawl? While there is no portion of the code whose specific purpose is to cause a denial-of-service of the machine, an educated guess can still be made. The first portion of the code that can slow systems down is the "fork" portion. The process of creating new processes very frequently is demanding and can be taxing on older systems, but that is not entirely

responsible for all the slow computers caused by this worm. Another thing this worm does is constantly reach out to other devices to establish a connection. At it's peak, this worm had infected 10% of the internet. That's 10% of all devices on a loop trying to connect to other devices running brute-force password attacks, which is very likely another reason no one was able to use their computer if this worm had installed itself. Lastly, there was the possibility that multiple worm processes could run on a single system. All of these factors combined paint a much clearer picture of why this worm caused so much chaos among those affected, and why this worm solidified itself as one of the most famous causing anywhere from $100,000 to $1 million dollars-worth of damages to many Ivy-League schools and government facilities.

Files

Name	# Lines	# Functions	Purpose
cracksome.c	443	8	Password cracker
hs.c	1367	34	Heuristic file
net.c	116	7	Network file
stubs.c	25	3	Unknown purpose
worm.c	174	4	Main file
worm.h	52	0	Object file
wormdes.c	758	6	DES encrypting passwords
x8113550.c	100	2	Unknown purpose
Total:	3035	64	

YARA Signatures

The next and final step in this process of analysis is to create signatures capable of detecting and identifying this worm, while also limiting false positives. These signatures will be compiled and run using YARA, and each will be composed of three parts: meta data, which will just give us information about the particular signature; strings, which will be plaintext or bytes that appear in the files; and conditions, which will define how the strings are used. In order to maximize the accuracy of identification (i.e., maximizing true positives and minimizing false positives), the parts of the files that should be used should fit one or both of two primary criteria. The first criterium is that the portion of the file we are identifying is unique to this worm, something that may fit this criterium would be certain function or variable names, strings that would be printed, or file paths. The second criterium is that the portion we are identifying is essential; any part of the program or file that would render the entire virus useless if removed would qualify here.

Figure 5.1 – "cracksome" YARA signature

```
private rule cracksome
{
        strings:
                $string_1 = "cracksome()"               // Name of main function
                $string_2 = "/etc/hosts.equiv"          // File path in strat_0
                $string_3 = "/.rhosts"                  // File path in strat_0
                $string_4 = "/.forward"                 // File path in strat_0
                $string_5 = "try_words()"               // Name of important function
                $string_6 = "dict_words()"              // Name of important function
                $string_7 = "/usr/dict/words"           // File path to large dictionary of words
                $string_8 = "#include \"worm.h\""       // Important file import

        condition:
                5 of them
}
```

This first signature was made for the “cracksome.c” file. A very brief description of why each string is included is to the right in the form of a comment, but I will also explain more explicitly why I have included most of them. The function names “cracksome”, “try_words”, and “dict_words” are all included because they have names that are not very common within programs (unlike “main” or “__init__”). The file paths are included because, while they do show up as strings within the program and therefore can be encoded to prevent detection, they are also otherwise required for the program to work due to either referencing an important local file from which data is pulled, or because they reference a critical part of a vulnerability. The last thing you will notice is the “include” statement. This is included because it refers to a file that is required to run this exploit, and most include or import statements cannot be encoded in any way.

Figure 5.2 – “hs” YARA signature

```
private rule hs
{
        strings:
                $string_1 = "#include <sys/socket.h>"    // Important library import
                $string_2 = "#include <netinet"          // Important library import
                $string_3 = "h_addr2host()"              // Important function structure import
                $string_4 = "h_name2host()"              // Important function structure import
                $string_5 = "try_rsh_and_mail("          // Name of important function
                $string_6 = "try_finger("                // Name of important function
                $string_7 = "getobjectbyname("           // Name of important function
                $string_8 = "xorbuff("                   // Name of important function
                $string_9 = "x50bc("                     // Name of important function
                $string_10 = "x488e("                    // Name of important function
                $string_11 = "exec /bin/sh"              // File path and execution
                $string_12 = "PATH=/bin:/usr/bin:/usr/ucb" // File path
                $string_13 = "gorch"                     // Important execution command
                $string_14 = "test_connection("          // Name of important function
                $string_15 = "#include \"worm.h\""       // Important file import

        condition:
                9 of them and filesize > 25KB
}
```

A few things to point out with the next signature, which was made for the “hs.c” file. The first string of note is string 2. Notice how it seems to end abruptly without a specific library or closing ‘>’. The reason for this is that in the code this import is referring to a subset of “netinet” called “in.h”, so the full import looks like “#include <netinet/in.h>”. I have left it unfinished because

both the previous string, as well as “#include <netinet>” would be valid imports and have no effect on running the worm overall. If I specified one or the other, I may miss the other valid import. Another string of interest is any of the important functions; notice how they all have an open parenthesis, but no closing parenthesis. The reason for this is to prevent any false positives of any of these strings showing up as a variable or comment, but to also be general enough to work even if the function has variables (without specifying what those variables are). The final area I want to focus on is the second condition included in this signature. Looking at the files table, hs.c is clearly much larger than any of the other files, so an additional constraint was added to account for this discrepancy (the actual file I have used for analysis was 32.2 kb, but there must be some room for difference).

The rest of the signatures look very similar to the two previous, and if you would like to look at all of them, the actual text file will be included with this report. That said, there is one last rule to tie the whole file together.

Figure 5.3 – master YARA signature

```
100 rule master
101 {
102         condition:
103                 5 of (cracksome, hs, net, stubs, worm, wormdes, x8113550)
104 }
```

This rule is in place to consider all the previous rules together. This allows the user to point this file at a folder containing all the relevant documents and get an overall conclusion on whether the Morris worm is in that folder or not. You may notice as well that only 5 of the 7 files need to match (there is also a similar pattern with previous signatures). The decision to not require all files to match is an important balance. If all 7 are required, you are far more likely to miss this virus by one of the files being changed or not included at all. If you only require a couple to match, you are far more likely to get a false positive, or a result that claims the virus is within the folder you targeted, even if it’s not. A decision should be made on an individual basis because there is no one right answer for all files.

Sources

FBI Article from 2018 - https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218

Radware Brief Description - https://www.radware.com/security/ddos-knowledge-center/ddospedia/morris-worm/

MIT Brief Description - https://groups.csail.mit.edu/mac/classes/6.805/articles/morris-worm.html

Cyberhoot Description and Video - https://cyberhoot.com/cybrary/morris-worm/

Metropolia Thesis - https://www.theseus.fi/bitstream/handle/10024/171871/bachlorThesis-Worm.pdf?sequence=2&isAllowed=y