The Internet Virus of November 3, 1988

Mark W. Eichin, MIT Project Athena

November 8. 1988

Contents

Abstract

This paper is a thorough analysis of the code of the virus program which attacked the Internet beginning some time November 3, 1988. It discusses the actual code itself, as well as the strategies and ideas involved in the propagation of the virus.

A virus, according to Webster's, is something which causes infectious disease, as well as being capable of growth and multiplication only in living cells. Inasmuch as a computer is analogous to a living entity, this program is a virus; one of its infection methods is very much like an actual virus, in that it actually infects a running program to gain entry into the system.

Also, virii infect. Worms just crawl around.

Chapter 1

Strategies Involved

1.1 Attacks

This virus attacked several things, directly and indirectly. It both picked out some deliberate targets and had interesting side effects.

1.1.1 Finger bug

The virus hit the finger daemon by overflowing a buffer which was allocated on the stack. The overflow was possible because a library function which did not do range checking was used. Since the buffer was on the stack. the overflow allowed a fake stack frame to be created: which caused a small piece of code to be executed when the procedure returned. The write daemon has a similar piece of code, which makes the same mistake. but it exec's `write` directly. and explicitly exits rather than returning. and thus never uses the (damaged] return stack.

1.1.2 Sendmail

The sendmail mechanism is the "debug" function, which enables debugging mode for the duration of the current connection. One thing that this enables is the ability to send a mail message with a piped program as the recipient. This mode is normally allowed in the sendmail configuration file or user .forward file directly: but not for incoming connections. In this case, the recipient was a command which would strip off the mail headers and pass the remainder of the message to a shell. The body was a script which created a C program which would suck over the rest of the modules from the host that sent it, and the commands to compile and execute it.

The fact that debug was enabled by default was reported to Berkeley by several sources during the 4.2 release, however it was not fixed for the 4.3 release (source or binary.) Project Athena was among a number of sites which disabled it, however it is unlikely that many binary-only sites were able to be as diligent.

1.1.3 rexec and passwords

The virus attacked by the Berkeley remote esecution protocol, which required the user name and plaintext password to be passed over the net. The program only used pairs of usernanies and

passwords which it. had already verified to be correct on the local host.

One fundamental security tenet violated here' was that passwords should not be at **all** readable by unprivileged entities. Under most forms of UNIX, the passwords are stored in as computationally extensive encryptions. However, this meant that a program needed merely ***try*** **a** large number of guesses, "on its own turf" so to speak, without going through any recorded channels. The Kerberos[2] system, used at Project Athena, keeps passwords only on a secure central machine, which is used as an authentication server. Although once a username was known, the password could be attacked in much the same way, the usernames are also stored centrally, making it more difficult for the virus to find **a** set of names to attack.

1.1.4 side effects

When it became clear that the virus was propagating via sendmail, the first reaction of many sites[3] was to cut **off** mail service. This did not totally stop the progress of the virus, which continued to travel via rexec and finger. It ***did*** effectivelv stop communication of information about the virus. slowing down the information about finger and the patches needed to fix the problem. USENET news was an effective side-channel of information spread. although **a** number of sites disabled that as well.

One program posted after the virus was analyzed was a tool to duplicate the password attack used (including the dictionary that the virus carried with it) to allow system administrators to analyze the passwords in use on their system. The spread of this virus should be effective in raising the awareness of users (and administrators] to the importance of choosing "difficult" passwords.

1.2 defenses

The virus used a number of techniques to hide itself as well, though they had various vulnerabilities.

1.2.1 covering tracks

The program did a number of things to cover its trail. It zeroed out its argument list, once it had finished processing the arguments, so that the process status command would not show how it was invoked.

It also deleted the executing binary. which would leave an inode only referenced by the execution of the program but not appearing in the filesystem. If the machine was rebooted while the virus was actually running, the file system salvager will recover the file after the reboot.

The program also uses resource limit functions to prevent it from using any space in a core dump. Thus, it prevents any bugs in the program from leaving core dumps behind.

[1] **ref orange book7**

[2] **cite paper**

[3] **including the Darpa** MILNET

1.2.2 camouflage

It was compiled as "sh", the same name used by the Bourne Shell, which is used often in shell scripts and automatic commands. Even a diligent system manager would probably not notice a large number of shells running for short periods of time. The virus did `fork`, splitting into a parent and child, approximately every three minutes. The parent would then die, leaving the child to continue from the exact same place.

1.3 flaws

The virus also had a number of flaws, varying between the subtle and the clumsy. Keith Bostic of Berkeley, with concurrence of the team at MIT[4] posted patches for some of the more obvious ones? as a humorous gesture.

1.3.1 reinfection prevention

The code for preventing reinfection of a machine which was actively infected didn't work at all. It was only checked on a one in fifteen random chance. making multiple infections likely. This also lead to the early detection of the virus, since only one in fifteen instances of the virus would actually die; since the virus was careful to clean up temporary files! its presence alone didn't interfere with reinfection.

Also, a multiply infected machine would spread the virus faster! perhaps proportionally to the number of infections it was harboring. since

- The program scrambles the lists of hosts and users it attacks; since the random number generator is seeded with the current time. the separate instances are likely to hit separate targets.

- The program tries to spend a large amount of time sleeping and listening for other infection attempts (which never report themselves) so that the processes would share the resources of the machine fairly well.

Thus, the virus spread much more quickly than the perpetrator expected, and was noticed for that very reason. The MIT Media Lab, for example, cut themselves completely off from the network because the computer resources absorbed by the virus were detracting from work in progress, while the lack of network service was a minor problem.

1.3.2 heuristics

One attempt that was made to make the program not waste time on non-UNIX systems was to first try to telnet to the host in the list. If the host refused telnet connections, it was likely to refuse other attacks as well. There were several problems with this attack:

[4] telephone call, around 9AM Friday?

- A number of machines exist which provide sendmail service (for example) that do not provide telnet service, and although vulnerable would be ignored under this attack.[5]

- The telnet "probing" code iminediately closed the connection upon finding that-it had opened it. By the time the "inet daemon", the funnel which handles most incoming network services, identified the connection and handed it off to the telnet daemon, the connection was already closed, causing the telnet daemon to indicate an error condition of high enough priority to get logged on most systems. Thus the times of the earliest attacks were noted, if not the machines they came from.

[5] ATHENA.MIT.EDU, for example. was vulnerable to the finger daemon attack. but was untouched because it did not run a telnet daemon.

Chapter 2

The program

2.1 main

The main module made several steps to set itself up.

2.1.1 initialization

First the program takes some steps to hide itself. It changes the "zeroth" argument, which is the process name, to "sh" so that no matter how the program was invoked, it would show up in the process table with the same name as the Bourne Shell. a program which is often running legitimately.

The program also sets the resource limit on core dump size to zero blocks. so that if the program did crash for some reason it would vanish. rather than leaving a core dump behind to help investigators. It also turns off handling of the write errors on pipes. which by default cause the program to **exit.**

The next step is to read the clock. store the current time in a local variable, and using that value to seed the random number generator.

2.1.2 Command line argument processing

The virus program itself takes an optional argument "-p" which must be followed by a decimal number, which seems to be a process id of the parent which spawned it. It uses this number later on to kill that process: probably to "close the door" behind it.

The rest of the command line arguments are "object names". These are names of files it tries to load in. If it can't load one of them, it quits. If the "-p" argument was given, it also deletes the file (and later tries to delete the running virus. as well as a file `/tmp/.dumb`.[1]

After **all** the arguments have been read. if no objects were loaded the program quits. It then checks for the existence of the object "11.c" and quits if it is missing.

If the "-p" argument was given, the program closes all of its file descriptors, and then deletes the files

The program then erases the text of all of the arguments.

[1] need better explanation of loadobject

It then scans **all** of the network interfaces on the machine, gets the flags and address of each interface. It tries to get the point to point address of the interface; it skips the loopback address. It **also** stores the netmask for that network.

Finally, it **kills off** the process **id** given with the "`-p`" option. It **also** changes **the** current process group, **so** that it doesn't die when the parent exits. Once this is cleaned up, it **falls** into the `doit` routine which performs the rest of the work.

2.2 doit routine

This routine is the where the program spends most of its time.

2.2.1 initialization

Like the main routine. it seeds the random number generator with the clock: and stores the clock value to later measure how long the virus has been running on this system.

It then tries `hg`. If that fails. it tries `hl`. If that fails. it tries `ha`.

It then tries to check if there is already a copy of the virus running on this machine. This code doesn't work correctly (one of the reasons the virus was using large amounts of computer time.)

It then sends a one byte on a TCP Stream connection to **128.32.137.13,** which **is** `ernie.berkeley.edu`. There has not been an explanation for this: it only **sends** this packet with a 1 in 15 random chance.

2.2.2 main loop

An infinite loop comprises the main active component of the virus. It calls the `cracksome` routine' which tries to find some hosts that it can break in to. Then it waits 30 seconds. while listening for bther virus programs attempting to break in. and tries to break into another batch of machines.

After this round of attacks. it forks. creating two copies of the virus; the original (parent) dies, leaving the fresh copy. The child copy has all of the information the parent had, while not having the accumulated CPU usage of the parent. It also has a new process id, making it hard to find.

The virus then runs the "h routines", which search for more machines to add to the list of hosts, and then sleeps for 2 minutes (again looking for other virus attempts.) After that, it checks to see if it has been running for more than 13 hours, and if so cleans up some of the entries in the host list.

Finally, before repeating, it checks `pleasequit`. If it is set, *and* it has tried more than 10 words from its own dictionary against existing passwords? it quits. Thus forcing `pleasequit` to be set in the system libraries will do very little to stem the progress of this virus.

2.3 Cracking routines

There are a collection of routines which are the "brain" of the virus. There is a **main** switch, which chooses which of four strategies to execute next, and a number of separate strategy routines. It is clearly the central point to add new strategies, were the virus to be further extended.

[2]This name **was** actually in the symbol table of the distributed binary.

2.3.1 cracksome

The `cracksome` routine is the main switch. Again, this routine was named in the global symbol table; though it could have been given a confusing or random `name`, it was actually clearly labelled, which lends some credence to the idea that the virus was released prematurely.

2.3.2 crack 0

The first crack routine read through the `/etc/hosts.equiv` file to find machine names that would be likely targets. While this file indicates what hosts the current machine trusts, it is fairly common to find systems where all machines in a cluster trust each other, and at very least implies that people with accounts on this machine will have accounts on the other machines mentioned in `hosts.equiv`.

It also read the `/.rhosts` file: which lists the set of machines that this machine trusts root access from. Note that it did not take advantage of any knowledge about this trust[3] but merely uses the names as a list of additional machines to attack. Often, system managers will deny read access to this file to any user other than root itself. to avoid providing any easy list of secondary targets that could be used to subvert the machine; this practice would also have prevented the virus from discovering those names, although `/.rhosts` is very often a subset of `/etc/hosts.equiv`.

The program then reads the entire local password file. `/etc/passwd`. It uses this to find persona! `.forward` files for names of other machines it can attack. It also records the username, encrypted password, and "gecos" information string which is also stored in the `/etc/passwd` file. After processing the entire file, it advances the attack type selector: so that the machine proceeds to the next set of attacks.

2.3.3 crack 1

The next set. of attacks are on passwords on the local machine. It uses several functions to pick passwords which can then be encrypted and matched against the encryptions obtained in phase 0:

- No password at all.
- The username itself.
- The username appended to itself.
- The second of the comma separated "gecos" information fields, which is commonly a nickname.
- The remainder of the full name after the first name in the "gecos" fields, ie. probably the last name, with the first letter converted to lower case.
- This "last name" reversed.

All of these attacks are applied to fifty passwords at a time from those collected in phase 0. If this pass finishes all of the passwords, it advances to phase 2.

[3] such as Bob Baldwin's system KUANG would

2.3.4 phase 2

Phase 2 takes the internal word list that the virus distributes with itself, and scrambles it. Then it takes the words one at a time and decodes them (the high bit is set on all of the characters to obscure them) and then tries them against. all collected passwords. Thus the check in-the main loop against `nextw` only succeeds after 10 of the words have been checked against all of the encryptions in the collected list.

Again, if the word list is exhausted the virus advances to phase 3.

2.3.5 phase 3

Phase 3 looks at the local `/usr/dict/words` file, a twenty four thousand word dictionary distributed with 4.3BSD and other unix systems. The words are stored in ?his file one word per line. One word at a time is tried against all encrypted passwords. If the word begins with an upper case letter. the letter is converted to lower case and the word is tried again.

When the dictionary runs out, the phase counter is again advanced to 4 (thus no more password cracking is attempted.)

2.4 h routines

The "h routines" are a collection of routines with short names, including hg, ha, hi, and hl, which search for other hosts to attack.

2.4.1 hg

The "hg" routine calls `rt_init` to scan the routing table, which creates a list of gateways. It then tries a generic attack routine[4] to attack via rsh, finger. and smtp.

2.4.2 ha

The "ha" routine also tries to go through the list of machines and connect to port 25, the SMTP port, to determine if a mailer was running on the machine.

2.4.3 hl

The "hl" routine just looks for certain machines based on their netmasks, and tries to attack them.

2.4.4 hi

The "hi" routine goes through the table of hosts and tries to actually attack a host via "rsh", "finger", "smtp".

[4] `s1638` internally

2.4.5 hul

The "hul" routine is called by the phase one and phase three crack subroutines. Once a user name and password is guessed, this routine is called with a hostname read from either the user's .forward or .rhosts files. It then runs an rsh to that machine, and has it execute a Bourne Shell, thus allowing it to use standard methods to attack that machine.

2.5 attack routines

There were a collection of attack routines, which all provided a Bourne Shell running on the remote machine if they succeeded.

2.5.1 hit finger

The "hit finger" routine tries to make a connection to the finger port of the remote machine. Then it creates a "magic packet" which consists of

- A 400 byte "runway" of VAX "nop" instructions, which can be executed harmlessly.
- A small piece of code which executes a Bourne Shell.
- A stack frame, with a return address which would hopefully point into the code.

Note that the piece of code is VAX code? and the stack frame is a VAX frame. in the wrong order for the Sun. Thus, although the Sun finger daemon has the same bug as the VAX one, this piece of code cannot exploit it.

The attack on the fingerd [5] can be considered a "viral" attack. since although the worm doesn't modify the host machine at all. the finger attack does modify the running finger daemon process. Then, speaking in viral terms, the "injected DNA" component of the virus contained the following VAX instructions:

```
pushl $68732f push '/sh<NUL>'
pushl $6e69622f push '/bin'
movl sp,r10 save address of start of string
pushl $0 push 0 (arg 3 to execve)
pushl $0 push 0 (arg 2 to execve)
pushl r10 push string addr (arg 1 to execve)
pushl $3 push argument count
movl sp,ap set argument pointer
chmk $3b do "execve" kernel call.
```

The execve system call causes the current process to be replaced with an invocation of the named program: /bin/sh is one of the UNIX command interpreters. In this case, the shell wound up running with its input coming from! and its output going to, the network connection. The virus then sent over the same bootstrap program that it used for its sendmail-based attack.

[5] William E. Sommerfeld, of MIT Project Athena. was the first to discover this mode of attack, and provided the description that follows.

2.5.2 hit rexec

The "hit rexec" routine uses the "exec/tcp" service, the remote execution system which is similar to rsh, but is designed for use by programs. It connects, sends the username, the password, and /bin/sh as the command to execute. It checks to see if it succeeded to connect and get access using one of the password/account pairs guessed earlier.

2.5.3 hit smtp

The "hit smtp" routine uses the "smtp/tcp" service to take advantage of the sendmail bug. It attempts to use the "debug" option to make sendmail run a command (the recipient of the message), which compiles a program (which is included as the body of the message.)

2.5.4 makemagic

This routine tries to make a telnet connection to the 6 addresses for the current victim, and then breaks it immediately. If it succeeds, it creates a listening stream socket on a random port number which the infected machine will eventually connect back to. Since it breaks the connection immediately, it often produces error reports from the telnet daemon, which get recorded, and provide some of the earliest reports of attack attempts.

2.6 host modules

There are a set of routines designed to collect names and addresses of target hosts in a master list.

2.6.1 name *to* host

This routine searches the host list for a given named host, returns the list entry describing it, and optionally adds it to the list if it isn't there already.

2.6.2 address to host

This routine searches the host list for a given host address, returns the list entry describing it, and optionally adds it to the list if it isn't there already.

2.6.3 add address

This routine adds an address to an entry in the host list. Each entry contains up to twelve names, up to six addresses, and a flag field.

2.6.4 add name

This routine adds a name to an entry in the host list, if it doesn't already exist.

2.6.5 clean up table

This routine cycles through the host list, and cleans out hosts which only have flag bits 1 and 2 set (and clears those bits.)

2.6.6 get addresses

This routine takes an element of the host table and tries to find an address for the name it has, or get a name for the addresses it has, and include the aliases it can find in the list as well.

2.7 object routines

These routines are what the system uses to pull all of its pieces into memory when it starts (after the host has been infected) and then to retrieve them to transmit to any host it infects.

2.7.1 load object

This routine opens the file, stats it, allocates enough space to load it in, reads it in as one block. decodes the block of memory (with XOR). If the object name contains a comma, it moves past it and starts the name there.

2.7.2 get object by name

This routine returns a pointer to the requested object. This is used to find the pieces to download when infecting another host.

2.8 other initialization routines

2.8.1 if init

This routine scans the array of network interfaces. It gets the flags for each interface, and makes sure the interface is UP and RUNNING (specific fields of the flag structure.) If the entry is a point to point type interface, the remote address is saved and added to the host table. It then tries to enter the router into the list of hosts to attack.

2.8.2 rt init

This routine runs "netstat -r -n" as a subprocess. This shows the routing tables, with the addresses listed numerically. It gives up after finding 500 gateways. It skips the default route, as well as the loopback entry. It checks for redundant entries, and checks to see if there this address is already an interface address. If not, it adds it to the list of gateways.

After the gateway list is collected, it scrambles it and enters the addresses in the host table.

Appendix A

Credits

I'd like to mention a few people who worked on the virus hunt:

A.1 The MIT team

Mark W. Eichin (Athena and SIPB) and Stanley R. Zanarotti (LCS and SIPB) lead the team disassembling the virus code. The team included William E. Sommerfeld (Athena, SIPB, and Apollo), Ted Y. Ts'o (Athena and SIPB), Jon Rochlis (MIT Telecom and SIPB): Hal Birkeland (MIT hledia Lab), and John T. Kohl (Xthena. DEC, and SIPB).

Jeffery I. Schiller (Director of the MIT Network. Athena. SIPB) did a lot of work in trapping the virus, setting up an isolated test suite, and dealing with the media. Ron Hoffman (MIT Telecom) was one of the first to notice an MIT machine attacked by finger.

Tim Shepard (LCS) provided information as to the propagtion of the virus, as well as large amounts of "netwatch" data and other technical help.

A.2 The Berkeley Team

We don't know how they were organized at Berkeley, however we conversed extensively and exchanged code with Keith Bostic throughout the morning of November 4, 1988.

A.3 Others

Numerous others across the country deserve thanks: many of them worked directly or indirectly on the virus, and helped coordinate the spread of information.