

Design and Implementation of a Simple Class Room Laboratory Internet Worm

Christopher Church, Tim Schmoyer, and Henry L. Owen*
School of Electrical and Computer Engineering
Georgia Institute of Technology; Atlanta, Georgia 30332-0250 USA
*Email: owen@ece.gatech.edu; Voice: 404-894-4126; fax 404-894-9959

Abstract

Beginners to Internet security research require an understanding of the inner workings of worms. A simple research worm designed to run on a test network facilitates this understanding. This paper explains the design and implementation of a simple research worm. A discussion of worm structure is provided. Design decisions made in creating our research worm are then given with an emphasis on how they relate to real Internet worms.

1 INTRODUCTION

Security researchers must possess an in-depth understanding of Internet worms in order to effectively combat them. Such an understanding best comes from seeing how a worm is designed and implemented. The aim of this paper is to provide insight into both the methods and mechanisms used in worm creation. First, a model will be defined for classifying worm-like code. Concrete details will then be explored through the design and implementation of a simple research worm. The worm described was created in a security research lab at the Georgia Institute of Technology to allow students to witness it spread in a safe, controlled environment. The information in this paper is being provided in hopes that it may help the reader defend against worm attacks by seeing a research worm from the worm author's viewpoint.

II. A GENERIC WORM MODEL

A general model of worm-like behavior is needed to understand and classify Internet worms. Ellis described six components of any worm system as: reconnaissance, a specific attack, a command interface, a communications capability, intelligence capabilities and unused attack capabilities [1]. Since the purpose of our educational worm is to demonstrate a worm's ability to infect and propagate hosts using a known vulnerability and exploit, rather than demonstrate an intelligent and cooperative worm system, we have chosen not to implement a command interface or maintain an intelligence capability in our prototype. In addition, maintaining unused attacks is unnecessary for our purpose. Nazario, et al., proposed a general worm algorithm that enumerates target hosts, verifies visibility, verifies vulnerability, exploits the vulnerability and then infects the target host [2]. Since we are using a specified vulnerability with a known exploit to reduce risk of harm to the network, we can combine the steps: verify visibility and verify vulnerability. We have added to the above algorithm the implementation of an optional payload. This can perform a number of functions including the communications and intelligence capabilities described earlier, as well as administrative or malicious operations on files and/or other resources on the infected host. Our choices provide a simplified general model to teach design decisions for our own worm. Our model is composed of five parts.

1. The enumeration of target hosts
2. The exploit of a weakness in security
3. The infection of a target
4. An optional payload
5. Virus-like reproduction

A. Enumeration of Target Hosts

Worms have used different techniques for discovering hosts to target on the network. These techniques can be classified in three categories: hit list, topological and scanning. A hit list is a pre-compiled list of known addresses for target hosts. A technique that mines information from the infected host for other targets uses a topological technique. The third technique is scanning an address set for target hosts.

Sequentially stepping through a subnet or using a pseudorandom number generator to scan through a large address space are methods used within this technique. These techniques can also be combined to increase the effective propagation rate of the worm through a network.

B. Exploit of a Vulnerability

All infections by an Internet worm begin with the exploit of a security weakness. This weakness must allow the parent worm to execute arbitrary code on the victim machine. The variety of vulnerabilities targeted by real worms is staggering. A detailed account of exploits used is beyond the scope of this paper; however several popular ones are worth mentioning. The ability for some email clients, such as Microsoft Outlook, to execute code attached to emails has remained a common point of attack. Worms have used email for years to spread rapidly. A recent worm to exploit this vulnerability is the SoBig.f email worm. Remote services with buffer overflow vulnerabilities are also commonly attacked. Buffer overflows have been the most commonly targeted [6] security vulnerability for years.

C. Infection

The infection stage begins when a host first receives and executes viral code. Typically, worms infect vulnerable hosts immediately after gaining access through a security hole. Some send intermediate code that downloads a copy of the worm and runs it [4]. The format of the code sent is dependant upon the type of exploit used and the type of system targeted. If the worm exploits multiple vulnerabilities, it may contain copies of its code in multiple formats. An extreme case of this is detailed in [5]. The exploit and infection code are the most common targets for signature based intrusion detection systems. These systems check packets on a network for known strings used in exploits or as parts of worms. Polymorphic code, or mutating the worm and exploit code randomly between each infection, may defeat these security systems. Few real worms have tried such techniques yet, but they should be expected [2].

D. Payload

All side effects other than reproduction and resource consumption are due to the payload of the worm. Interesting examples from existing worms include deletion of files, inclusion of files into the worm itself, installation of backdoors, denial of service attacks, installation of security patches, password cracking, spamming, and computer restarts [7].

E. Viral Reproduction

All worms must infect new targets in order to reproduce. The vulnerabilities used will determine exactly how this is done. Internet worms may be described as either active or passive. Active worms have an autonomous cycle; they spread without human interaction. Passive worms require some interaction in order to infect a host. Common passive worm examples include email worms that infect when a user opens an attachment.

F. Summary of Worm Model

A five-part worm model makes it easy to analyze existing worms in terms of their subcomponents. Our worm model will be used to describe the construction of a research worm and compare it to worms found on the Internet.

III. DESIGN OF THE SPOC WORM

The design of the worm, affectionately dubbed the Simple Proof Of Concept (SPOC) worm tries to attain generality, simplicity, and usefulness. SPOC must be as simple as possible. However, the very nature of worms is complex, so even the most basic implementation will reflect this to an extent.

A. Platform

The Intel x86 architecture is an obvious choice for SPOC so that it could be run in a Georgia Tech security research lab. Most malicious worms target Intel machines, so making SPOC do the same also assists in studying common behavior. The choice of operating system is slightly more difficult. Both GNU/Linux and Microsoft Windows are common targets for Internet worms. Windows has received slightly more

worm attacks lately, but Linux has a better collection of free development tools. Linux was chosen mainly to make development and analysis easier.

B. Vulnerability

SPOC targets a buffer overflow in a mock network service [3]. A buffer overflow occurs when a data buffer is written with more data than it can hold and some of the data overwrites other variables. They are commonly exploited in buffers located on the stack because the address where functions return is located nearby. Buffer overflows are the most common vulnerabilities found in computers on the Internet. The implementation of the fake service contains software checks to monitor the worm's progress. SPOC doesn't attack a real service; therefore it is harmless if ever run on a network connected to the Internet. The implementation of the vulnerable service is only about 150 lines of C code. It implements an echo service that listens on port 3333. The critical vulnerability occurs within the code shown in figure 1.

```

void svcHandle(int sockfd)
{
    char userinput[16];

    dup2(sockfd, STDIN,
        FILENO);

```

Figure 1: vulnerable portion of service

The memory layout of the vulnerable service can be seen in figure 2. When *svcHandle()* finishes executing it jumps to the return address stored on the stack. The address to return to was placed there when *svcHandle()* was called. If a string larger than 16 bytes is sent to *userinput* then part of the data will overwrite the return address. Through trial and error we can guess a reasonable stack address, send a string of NOP (No Operation) instructions, and a string of code to execute. The NOP string allows some degree of error to our guess; it will work as long as the return address points somewhere in the NOP area. So our malicious code is laid out as shown in figure 3.

16 bytes	...	Higher Addresses
	userinput	
4 bytes	return address	Lower Addresses
	...	

Figure 2: Stack layout in vulnerable service

16 bytes	4 bytes	500 bytes	7 bytes
padding	stack address	NOPs	code

Figure 3: Layout of malicious code

C. Payload

There is no payload in SPOC, although it is designed to be easy to add one. Useful payloads to consider adding in the future include sending messages to the system log, restarting the vulnerable service, or sending an email notice of success.

D. Reproduction

The method for determining which IP addresses to attack is a difficult decision for worm designers. Worm authors will try to make the address generation seem random in order to slow down Internet security defenses. They want to scatter addresses among many different networks to spread their worm faster. They will also try to prevent different instances of the worm from repeating attacks on the same addresses. The reproduction strategy used by SPOC aims to balance simplicity with infection speed. SPOC first attempts to infect every machine on the subnet of each network interface. This technique, very similar to the strategy used by Code Red 2 [7], spreads the worm quickly throughout local networks. After attacking each subnet, a random IP address is set as the target address. SPOC enters a loop in which the target address is attacked and incremented. By starting at a random address, different copies of the worm are likely to attack different machines. This strategy has the added benefit of not requiring synchronization between the worms.

IV. IMPLEMENTATION

The construction process involves a sequence of stages that result in attack code, which may be used in a buffer overflow attack. A high-level pseudo-code description is given first, followed by a C implementation of the majority of SPOC, and finally a complete worm is written in assembly. The assembly is transformed into a string that can be embedded in a C program. Each stage allows us to focus on a few details of the implementation at a time. Implementing SPOC in both C and assembly may seem unnecessary at first. However, both have drawbacks that make neither a reasonable choice as the only implementation language. The code generated by a C compiler implements function calls and variable references in a way that cannot be used in a buffer overflow attack. Assembly allows us to solve these problems but makes it difficult to write non-trivial code. The problems of both may be avoided by writing the entire worm in C except the code that builds the attack string, and then modifying the assembly produced by a C compiler.

A. Pseudo-Code

The construction process begins with pseudo-code. At this level, ideas of SPOC's behavior may be discussed without worrying too much about implementation details. Since SPOC is an active worm, the bulk of its time is spent running a loop that scans for targets and tries to infect them. Even though many worms achieve great speedups by running multiple threads, SPOC is kept simple and uses only one thread of execution. Each loop iteration begins by computing a new target address. A connection is attempted to the target address. If the connection fails, SPOC gives up and moves on to the next address. If the connection opens successfully, a specially crafted attack string is sent and the connection is closed. Stealthier schemes could be devised where the victim is probed first to see if it is vulnerable. However, most Internet worms don't do this and neither does SPOC. Based upon the above description, the pseudo-code in figure 4 is easily devised. Written in very informal notation, it gives an excellent starting point for the implementation of SPOC.

B. The C Program

The next step in the construction process is to write C code based on the pseudo-code description. This is easy since the pseudo-code is extremely similar in structure to C code. The important additions at this stage are system and library calls.

1. Library Calls

The network interfaces must be discovered before SPOC can attack machines connected to them. Linux systems allow for a list of local interfaces to be obtained with the *ioctl()* system call. To obtain the listing, the second argument to *ioctl()* must be *SIOCGIFCONF*. The exact usage is well documented in [8]. The connect, send, and close operations need to be performed as well. The most common network library used in Linux is the BSD sockets library. A TCP connection to the vulnerable server is opened with a call to *connect()*, data sent with *send()*, and the connection terminated with *close*. A detailed explanation of the sockets library may be found in [9].

```

procedure attack (address)
begin
  connect to address, port 3333;

  if connection succeeds then
  begin
    send vuln_code;
    close connection;
  end
end

procedure worm_main
begin

  for each network_interface do
  for each address on
  network_interface subnet do
    attack (address);

  random_address = rand_32bits;

  loop forever
  begin
    attack (random_address);
    increment (random_address);
  end
end

```

Figure 4: pseudo-code for SPOC

2. Implementation

We have enough information to rewrite the pseudo-code in C. Details of conversion of the pseudo code to C code are omitted.

3. Testing

We wanted to be able to test this code after it was developed. The *code* variable, which holds the code to execute on a victim machine, was initialized to contain code to generate a UDP packet. The packet is sent back to the attacker and contains the string “hello.” Another program running on the attacker machine receives the packet and announces that an attack has succeeded. This setup made it clear whether the vulnerable service was being exploited correctly.

C. Assembly Language

In the assembly language stage, several important problems relating to use of libraries, memory addressing, and self-reference must be resolved. The C code might seem to complete the worm, as if SPOC would be finished if the *code* variable pointed to the memory location of the beginning of *main()*. There are several problems that are not apparent until the assembly generated by the compiler is carefully analyzed.

1. Function Calls

A commonly available tool on GNU/Linux systems for analyzing executable files is the GNU Debugger (gdb) [10]. In order to use it, we must compile the C source in gcc with the *-g* and *-static* options. We can look at the assembly code for any function in gdb by using the *disassemble* command. This is especially useful, because it allows us to see the code behind library functions. When either of the two functions shown in the C code is disassembled, a sequence of assembly instructions appears. Wherever a library call was made it has been replaced by

```
call <memory location> <function>
```

It is unknown where the attack code will be in memory when it runs, so any library function calls will point to an incorrect addresses. A solution is to replace procedure calls with the code they contain, known as inline expansion. For example, instead of

```
call 0x804d180 <__ioctl>
```

one could write

```
push  %ebx
mov   0x10(%esp,1),%edx
mov   0xc(%esp,1),%ecx
mov   0x8(%esp,1),%ebx
mov   $0x36,%eax
int   $0x80
pop   %ebx
```

which is what appears from the command *disassembleioctl* in *gdb*. This technique works rather well for functions that wrap system calls. *write()*, *ioctl()*, *send()*, and *close()* are such functions. This quickly becomes unreasonable with functions like *printf()* that contain several lengthy subroutines. Direct system calls are used by a worm author whenever possible.

2. Memory References

Another problem occurs whenever memory is referenced. Many system calls take pointers as arguments. Allocating and writing the data in these pieces of memory is non-trivial. A similar problem occurs when we wish to send a copy of the worm to a victim; we do not know where in memory the worm code begins. An interesting trick has been used by exploit writers for years to resolve these problems. The *call* instruction jumps and pushes the address of the next instruction on the stack. This may be exploited to obtain the address of the worm and of a data storage region as follows.

```
WORM:
call TOP
TOP:
pop %ebp
jmp BOTTOM_CALL

BODY:
pop %esi
...
BOTTOM_CALL:
call BODY
```

Register *EBP* contains the address of the beginning of the block of code and register *ESI* contains the address of the byte immediately following the code. *ESI* may be used as the beginning of an area that is safe to use for storing data. The size of this area is equal to the size of the vulnerable service's stack minus the size of the worm. Considering that it is only necessary to store *socketaddr* and *ifconf* structures, this should be more than adequate.

3. Summary of Assembly Conversion

Inline expansion and abuse of the *call* instruction solve the big problems involved with writing attack code. It is now possible to calculate the address of the code, find an area of memory that may be used for data storage, and flatten functions so that they can be used. Writing the assembly code is now a straightforward task.

D. Hex String

The worm code is almost finished and the only step remaining is to convert it into a string of hexadecimal characters. This string may be used to initialize the *code* variable in the C program. The conversion is easily done with gdb's *x/bx* command. This command gives hexadecimal byte values for regions of an executable. One real problem remains: removing any input-terminating bytes from the string. Buffer overflow vulnerabilities are often found in functions, which read a string of data until a terminating character occurs. Common terminating characters are null characters and new lines. The *gets()* function SPOC exploits functions by reading until a new line is encountered. This is bad if a new line occurs in the middle of the malicious string! The easiest way to handle this is to rewrite assembly instructions so that there are no terminating characters. For example, the sequence

```
mov %ebp, 0
```

could remove the zero value by writing it as

```
xor %ebp, %ebp
```

V. TESTING THE WORM

The network it was run on at Georgia Tech was isolated from the Internet and from other systems. Several computers ran copies of the vulnerable service while another launched the worm. Worm progress was easily monitored with two utilities: *ethereal* and *strace*.

Ethereal is a Linux based packet sniffer that was used to watch for traffic across the vulnerable service port. Each copy of the vulnerable service was launched using *strace*. *Strace* is another Linux program that prints a log of every system call made. Since network connections are opened and packets are sent via system calls, the log produced by *strace* shows what the mock network service is doing.

```
socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP) = 4

connect(4, {sa_family=AF_INET,
sin_port=htons(3333),

sin_addr=inet_addr("192.168.0.1")},
16) = -1 ECONNREFUSED (Connection
refused)

close(4)
= 0

socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP) = 4

connect(4, {sa_family=AF_INET,
sin_port=htons(3333),
sin_addr=inet_addr("192.168.0.2")},
16) = -1 ECONNREFUSED (Connection
refused)

close(4)
= 0
```

Figure 5: Strace output of normal behavior

When the vulnerable service executes it must bind to a port, listen on that port until a connection is attempted, and accept a connection when it occurs. A successful connection is handled by creating a new

process to handle the transaction. Each step mentioned above is performed with a system call and is printed to the console by strace when it occurs. Figure 5 shows a sample of the output for this portion of the program. The process spawned for each transaction will read a string from the network, echo it back out the network connection, and terminate both the connection and the process. Strace reports each of these actions as they occur. When the SPOC Worm infects the mock network service, a distinct pattern of system calls appears in the output from strace. This signature pattern appears as *socket()*, *connect()*, and *close()* system calls being executed very quickly. A sample of the output is shown in figure 6.

```
bind(3, {sa_family=0x200 /*
AF_??? */ ,
sa_data="\r\5\0\0\0\0\0\0\0\0\0\0\0"}, 16) = 0

listen(3, 128)
= 0

accept(3, {sa_family=AF_INET,
sin_port=htons(32796),
sin_addr=inet_addr("127.0.0.1"
)}, [16]) = 4
```

Figure 6: Strace output of worm behavior

The worm repeatedly tries to open connections to port 3333 (the port that the vulnerable service runs on). Machines located on the same LAN are tried first, in this case 192.168.0.1 and 192.168.0.2. As described in section 4, if the worm connects successfully it sends a special string formatted to overflow the buffer and execute a copy of the worm. This attack would appear in the strace output as a sequence of sends.

After each infection attempt the worm continues searching for vulnerable services. Random IP addresses are generated and attacked after all machines on the LAN have been scanned. Upon infection, the victim machine will begin displaying the pattern of system calls that show the worm scanning for more targets. Worm behavior is easy to monitor with utilities like strace.

VI. CONCLUSION

The creation of the SPOC worm provides a working worm to experiment with in a test environment. This research worm allows the safe examination and exploration of the worm process, including the authoring process. The construction process itself provides valuable insight into the techniques used by malicious authors. Examination of some of the tradeoffs and authoring techniques yields a better understanding of worms, allowing better defensive techniques to be developed.

Another valid and useful approach to worm research is of course to analyze worms that were captured from the Internet. Our work took a different approach by attempting to create a simple worm that may be experimented with in a laboratory environment. By creating our own vulnerable service, our research worm is harmless should it be released outside the laboratory environment. We think our approach with a simple, relatively easy to understand authoring process as well as simple result is another valid approach to teaching worm analysis.

VII. REFERENCES

[1] Dan Ellis, Worm Anatomy and Model, Proceedings of the 2003 ACM Workshop, October 2003, pp. 42-50, <http://mason.gmu.edu/~esibley/INFS697F03/ACM%20Worm%20Anatomy%20and%20Model.pdf>

- [2] Jose Nazario, Jeremy Anderson, Rick Wash and Chris Connelly, The Future of Internet Worms, Crime labs research, July 2001, http://www.cgisecurity.com/lib/the_future_of_internet_worms.pdf
- [3] Cowan et al. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." Proceedings of DARPA Information Survivability Conference and Expo (DISCEX). Oregon Graduate Institute of Science and Technology, 1999.
- [4] Eugene Spafford, "An Analysis of the Internet Worm," Proc. European Software Engineering Conference, pp. 446-468, Sept, 1989. Lecture Notes in Computer Science #387, Springer-Verlag.
- [5] Wiley, Brandon. "Curious Yellow: The First Coordinated Worm Design." http://blanu.net/curious_yellow.html
- [6] http://www.phrack.org/phrack/61/p61-0x09_Polymorphic_Shellcode_Engine.txt, "Polymorphic Shellcode Engine using Spectrum Analysis."
- [7] Weaver, Nicholas, "A Brief History of the Worm." Security Focus: INFOCUS, Nov. 26, 2001, <http://www.securityfocus.com/infocus/1515>
- [8] <http://www.phrack.org/phrack/49/P49-14>, "Smashing the Stack for Fun and Profit."
- [9] Stevens, W. Richard Unix Network Programming, Volume 1. Prentice Hall, 1998.
- [10] Donahoo, Michael J. and Calvert, Kenneth L. The Pocket Guide to TCP/IP Sockets. San Francisco: Morgan Kaufmann, 2001.
- [11] <http://www.gnu.org/manual/gdb/>, "Debugging with GDB."
- [12] Zalewski, Michael. "I don't think I really love you." <http://commons.somewhere.com/rre/2000/RRE.worm.design.html>