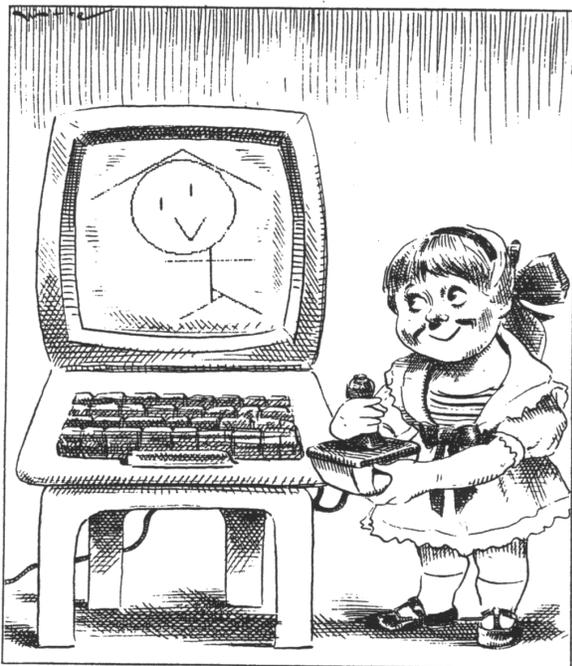


## 17:09 Protecting ELF Files by Infecting Them

by Leandro “acida” Pereira

Writing viruses is a sure way to learn not only the intricacies of linkers and loaders, but also techniques to covertly add additional code to an existing executable. Using such clever techniques to wreck havoc is not very neighborly, so here’s a way to have some fun, by injecting additional code to tighten the security of an ELF executable.

Since there’s no need for us to hide the payload, the injection technique used here is pretty rudimentary. We find some empty space in a text segment, divert the entry point to that space, run a bit of code, then execute the program as usual. Our payload will not delete files, scan the network for vulnerabilities, self-replicate, or anything nefarious; rather, it will use `seccomp-bpf` to limit the system calls a process can invoke.



### Caveats

By design, `seccomp-bpf` is unable to read memory; this means that string arguments, such as in the `open()` syscall, cannot be verified. It would otherwise be a race condition, as memory could be modified after the filter had approved the system call dispatch, thwarting the mechanism.

It’s not always easy to determine which system calls a program will invoke. One could run it under `strace(1)`, but that would require a rather high test coverage to be accurate. It’s also likely that the standard library might change the set of system calls, even as the program’s local code is unchanged. Grouping system calls by functionality sets might be a practical way to build the white list.

Which system calls a process invokes might change depending on program state. For instance, during initialization, it is acceptable for a program to open and read files; it might not be so after the initialization is complete.

Also, `seccomp-bpf` filters are limited in size. This makes it more difficult to provide fine-grained filters, although eBPF maps<sup>31</sup> could be used to shrink this PoC so slightly better filters could be created.

### Scripting like a kid

Filters for `seccomp-bpf` are installed using the `prctl(2)` system call. In order for the filter to be effective, two calls are necessary. The first call will forbid changes to the filter during execution, while the second will actually install it.

The first call is simple enough, as it only has numeric arguments. The second call, which contains the BPF program itself, is slightly trickier. It’s not possible to know, beforehand, where the BPF program will land in memory. This is not such a big issue, though; the common trick is to read the stack, knowing that the `call` instruction on x86 will store the return address on the stack. If the BPF program is right after the `call` instruction, it’s easy to obtain its address from the stack.

<sup>31</sup>man 2 bpf

```

1   ; ...
3   jmp filter
5 apply_filter:
   ; rdx contains the addr of the BPF program
7   pop rdx
9   ; ...
11  ; 32bit JMP placeholder to the entry point
   db 0xe9
13  dd 0x00000000
15 filter:
   call apply_filter
17 bpf:
19  bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 4
   ; remainder of the BPF payload

```

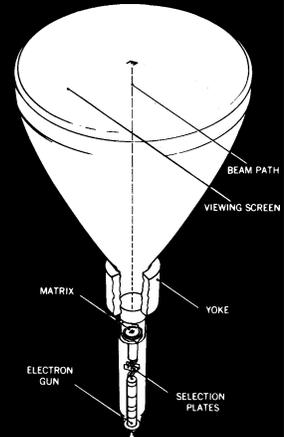
The BPF virtual machine has its own instruction set. Since the shell code is written in assembly, it's easier to just define some macros for each BPF bytecode instruction and use them.

```

bpf_ld   equ 0x00
2 bpf_w   equ 0x00
bpf_abs  equ 0x20
4 bpf_jump equ 0x05
bpf_jeq  equ 0x10
6 bpf_k    equ 0x00
bpf_ret  equ 0x06
8
seccomp_ret_allow equ 0x7fff0000
10 seccomp_ret_trap equ 0x00030000
audit_arch_x86_64 equ 0xc000003e
12
%macro bpf_stmt 2 ; BPF statement
14   dw (%1)
   db (0)
16   db (0)
   dd (%2)
18 %endmacro
20
%macro bpf_jump 4 ; BPF jump
   dw (%1)
22   db (%2)
   db (%3)
24   dd (%4)
%endmacro
26
%macro sc_allow 1 ; Allow syscall
28   bpf_jump {bpf_jump+bpf_jeq+bpf_k}, 0, 1, %1
   bpf_stmt {bpf_ret+bpf_k}, seccomp_ret_allow
30 %endmacro

```

## CHARACTRON® SHAPED BEAM TUBES



Information is displayed on tube screens ranging from 5" to 21" in diameter. Many of these tubes used in the SAGE system achieved 20,000 hours or more of reliable performance.

Heart of the CHARACTRON Tube is a stencil-like matrix, a tiny disc with alphanumeric and symbolic characters etched through it. The matrix is placed within tube neck, in front of an electron gun.

The electron stream is extruded through a selected character in the matrix, forming the beam into the desired character shape. When the beam impinges on the phosphor-coated tube face, the character is reproduced. In compact tubes the entire matrix is flooded with electrons, generating a complete array of characters. Only the desired character is allowed to pass through a masking aperture. By actually forming the character or symbol from the electron beam, the tube provides the highest available definition of character generation and overall display quality.

**SPEED UP & STREAMLINE CALCULATIONS!**

# USE **ADDIMAX**

the *ONE* and *ONLY* pocket adding machine with a *CREDIT BALANCE!*

**SECTION THRU DROP PANEL**

**POSITION OF THE C. OF G.**

A	y	A·y	
13.12.75	= 1170	375	4400
12.11	= 1450	55	8000
$\Sigma A$	= 2620	$\Sigma A \cdot y$	= 12400

$y_c = \frac{12400}{2620} = 4.74$

$I = \frac{156}{3} \left| \frac{4.74}{3} + 2.76 \right|$

$= \frac{156}{3} \left| 95.1 + 22.1 \right| = 6700 \text{ inch}^4$

**credit balance shown here**  
 ( 12.85 - 20.00 = -7.15 )

**only \$4.95**  
 genuine leather case incl.

**ADDIMAX ORIGINAL-ADDIATOR**

**SUB ZERO BALANCE**

24. (Actual size)

**Only ADDIMAX has two rows of answer windows. Look for them in a machine you buy!**

By listing all the available system calls from `syscall.h`,<sup>32</sup> it's trivial to write a BPF filter that will deny the execution of all system calls, except for a chosen few.

```

1 bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 4
2 bpf_jump {bpf_jmp+bpf_jeq+bpf_k}, 0, 1,
   audit_arch_x86_64
3 bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 0
4 sc_allow 0 ; read(2)
5 sc_allow 1 ; write(2)
6 sc_allow 2 ; open(2)
7 sc_allow 3 ; close(2)
8 sc_allow 5 ; fstat(2)
9 sc_allow 9 ; mmap(2)
10 sc_allow 10 ; mprotect(2)
11 sc_allow 11 ; munmap(2)
12 sc_allow 12 ; brk(2)
13 sc_allow 21 ; access(2)
14 sc_allow 158 ; prctl(2)
15 bpf_stmt {bpf_ret+bpf_k}, seccomp_ret_trap

```

## Infecting

One of the nice things about open source being ubiquitous today is that it's possible to find source code for the most unusual things. This is the case of `ELFKickers`, a package that contains a bunch of little utilities to manipulate ELF files.<sup>33</sup>

I've modified the `infect.c` program from that collection ever so slightly, so that the placeholder `jmp` instruction is patched in the payload and the entry point is correctly calculated for this kind of payload.

A `Makefile` takes care of assembling the payload, formatting it in a way that it can be included in the C source, building a simple guinea pig program twice, then infecting one of the executables. Complete source code is available.<sup>34</sup>

```

1 #include <stdio.h>
2 #include <sys/socket.h>
3
4 int main(int argc, char *argv[]) {
5     if (argc < 2) {
6         printf("no socket created\n");
7     } else {
8         int fd=socket(AF_INET, SOCK_STREAM, 6);
9         printf("created socket, fd = %d\n", fd);
10    }
11 }

```

## Testing & Conclusion

The output in Figure 22 is an excerpt of a system call trace, from the moment that the `seccomp-bpf` filter is installed, to the moment the process is killed by the kernel with a `SIGSYS` signal.

*Happy hacking!*

<sup>32</sup>`echo "#include <sys/syscall.h>" | cpp -dM | grep '^#define __NR_'`

<sup>33</sup>`git clone https://github.com/BR903/ELFKickers || unzip pocorgtfo17.pdf ELFKickers-3.1.tar.gz`

<sup>34</sup>`unzip pocorgtfo17.pdf infect.zip`

```

1 prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) = 0
2 prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, {len=30, filter=0x400824}) = 0
3 socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 41
4 --- SIGSYS {si_signo=SIGSYS, si_code=SYS_SECCOMP, si_call_addr=0x7f2d01aa19e7,
5     si_syscall=__NR_socket, si_arch=AUDIT_ARCH_X86_64} ---
6 +++ killed by SIGSYS (core dumped) +++
7 [1] 27536 invalid system call (core dumped) strace ./hello

```

Figure 22. Excerpt of `strace(1)` output when running `hello.c`.

## 17:10 Laphroaig's Home for Unwanted Polyglots and Oday

*from the desk of Pastor Manul Laphroaig,  
Tract Association of PoC||GTFO.*

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.



Now it's your turn to share what you know, that nifty little truth that other folks might not yet know. It could be simple, or a bit advanced. Whatever your nifty tricks, if they are clever, we would like to publish them.

Do this: write an email in 7-bit ASCII telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick explanation would do.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal language and hand-drawn diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to [pastor@phrack.org](mailto:pastor@phrack.org) and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middleing our submission process.

Yours in PoC and Pwnage,  
Pastor Manul Laphroaig, T.G. S.B.

### **PATENTS**

**I**F YOU HAVE AN INVENTION which you wish to patent you can write fully and freely to Munn & Co. for advice in regard to the best way of obtaining protection. Please send sketches or a model of your invention, and a description of the device, explaining its operation.

All communications are strictly confidential. Our vast practice, extending over a period of seventy years, enables us in many cases to advise in regard to patentability without any expense to the client. Our Handbook on Patents is sent free on request. This explains our methods, terms, etc., in regard to **Patents, Trade Marks, Foreign Patents, etc.**

All patents secured through us are described without cost to the patentee in the *Scientific American*.

## **MUNN & CO.**

*SOLICITORS OF PATENTS*

**699 WOOLWORTH BLDG., NEW YORK  
and 625 F STREET, WASHINGTON, D. C.**