7 A Ghetto Implementation of CFI on x86

by Jeffrey Crowell

In 2005, M. Abadi and his gang presented a nifty trick to prevent control flow hijacking, called *Control Flow Integrity*. CFI is, essentially, a security policy that forces the software to follow a predetermined control flow graph (CFG), drastically restricting the available gadgets for return-oriented programming and other nifty exploit tricks.

Unfortunately, the current implementations in both Microsoft's Visual C++ and LLVM's clang compilers require source to be compiled with special flags to add CFG checking. This is sufficient when new software is created with the option of added security flags, but we do not always have such luxury. When dealing with third party binaries, or legacy applications that do not compile with modern compilers, it is not possible to insert these compile-time protections.

Luckily, we can combine static analysis with binary patching to add an equivalent level of protection to our binaries. In this article, I explain the theory of CFI, with specific examples for patching x86 32-bit ELF binaries—without the source code.

CFI is a way of enforcing that the intended control flow graph is not broken, that code always takes intended paths. In its simplest applications, we check that functions are always called by their intended parents. It sounds simple in theory, but in application it can get gnarly. For example, consider:

```c
int a() { return 0; }
int b() { return a(); }
int c() { return a() + b() + 1; }
```

For the above code, our pseudo-CFI might look like the following, where `called_by_x` checks the return address.

```c
int a() {
  if (!called_by_b && !called_by_c) {
    exit();
  }
  return 0;
}
int b() {
  if (!called_by_c) {
    exit();
  }
  return a();
}
int c() { return a() + b() + 1; }
```

Of course, this sounds quite easy, so let's dig in a bit further. Here is a very simple example program to illustrate ROP, which we will be able to effectively kill with our ghetto trick.

```c
#include <string.h>

void smashme(char* blah) {
  char smash [16];
  strcpy(smash, blah);
}

int main(int argc, char** argv) {
  if (argc > 1) {
    smashme(argv[1]);
  }
}
```

In x86, the stack has a layout like the following.

Local Variables
Saved `ebp`
Return Pointer
Parameters
...

By providing enough characters to `smashme`, we can overwrite the return pointer. Assume for now, that we know where we are allowed to return to. We can then provide a whitelist and know where it is safe to return to in keeping the control flow graph of the program valid.

Figure 4 shows the disassembly of `smashme()` and `main()`, having been compiled by GCC.

Great. Using our whitelist, we know that `smashme` should only return to `0x08048456`, because it is the next instruction after the `ret`. In x86, `ret` is equivalent to something like the following. (This is not safe for multi-threaded operations but we can ignore that for now.)

```
pop ecx ; puts the return address to ecx
jmp ecx ; jumps to the return address
```

```
[0x08048320]> pdf@sym.smashme
/ (fcn) sym.smashme 26
|          ; arg int arg_2        @ ebp+0x8
|          ; var int local_6      @ ebp-0x18
|          ; CALL XREF from 0x08048451 (sym.smashme)
|          0x0804841d      55              push ebp
|          0x0804841e      89e5            mov ebp, esp
|          0x08048420      83ec28          sub esp, 0x28
|          0x08048423      8b4508          mov eax, dword [ebp+arg_2]   ; [0x8:4]=0
|          0x08048426      89442404        mov dword [esp + 4], eax
|          0x0804842a      8d45e8          lea eax, [ebp-local_6]
|          0x0804842d      890424          mov dword [esp], eax
|          0x08048430      e8bbfeffff      call sym.imp.strcpy
|          0x08048435      c9              leave
\          0x08048436      c3              ret
[0x08048320]> pdf@sym.main
/ (fcn) sym.main 33
|          ; arg int arg_0_1      @ ebp+0x1
|          ; arg int arg_3        @ ebp+0xc
|          ; DATA XREF from 0x08048337 (sym.main)
|          ;-- main:
|          0x08048437      55              push ebp
|          0x08048438      89e5            mov ebp, esp
|          0x0804843a      83e4f0          and esp, 0xfffffff0
|          0x0804843d      83ec10          sub esp, 0x10
|          0x08048440      837d0801        cmp dword [ebp + 8], 1       ; [0x1:4]=0x1464c45
|      ,=< 0x08048444      7e10            jle 0x8048456
|      |   0x08048446      8b450c          mov eax, dword [ebp+arg_3]   ; [0xc:4]=0
|      |   0x08048449      83c004          add eax, 4
|      |   0x0804844c      8b00            mov eax, dword [eax]
|      |   0x0804844e      890424          mov dword [esp], eax
|      |   0x08048451      e8c7ffffff      call sym.smashme
|      |   ; JMP XREF from 0x08048444 (sym.main)
|      `-> 0x08048456      c9              leave
\          0x08048457      c3              ret
```

Figure 4 – Disassembly of `main()` and `smashme()`.

Cool. We can just add a check here. Perhaps something like this?

```
pop ecx; puts the return address to ecx
cmp ecx, 0x08048456; check that we return to the right place
jne 0x41414141; crash
jmp ecx; effectively return
```

Now just replace our `ret` instruction with the check. `ret` in x86 is simply this:

```
$ rasm2 -a x86 -b32 "ret"
c3
```

where our code is this:

```
$ rasm2 -a x86 -b32 "pop ecx;cmp ecx, 0x08048456; jne 0x41414141; jmp ecx"
5981f9568404080f8534414141ffe1
```

Sadly, this will not work for several reasons. The most glaring problem is that `ret` is only one byte, whereas our fancy checker is 15 bytes. For more complicated programs, our checker could be even larger! Thus, we cannot simply replace the `ret` with our code, as it will overwrite some code after it—in fact, it would overwrite`main`. We'll need to do some digging and replace our lengthy code with some relocated parasite, symbiont, code cave, hook, or detour—or whatever you like to call it!

Nowadays there aren't many places to put our code. Before x86 got its no-execute (NX) MMU bit, it'd be easy to just write our code into a section like `.data`, but marking this as `+x` is now a huge security hole, as it will then be `rwx`, giving attackers a great place for putting shellcode. The `.text` section, where the main code usually goes, is marked `r-x`, but there's rarely slack space enough in this section for our code.

Luckily, it's possible to add or resize ELF sections, and there're various tools to do it, such as *Elfsh*, *ERESI*, etc. The challenge is rewriting the appropriate pointers to other sections; a dedicated tool for this will be released soon. Now we can add a new section that is marked as `r-x`, replace our `ret` with a jump to our new section—and we're ready to take off!

Well, wheels aren't up yet. As mentioned before, `ret` is `c3`, but absolute jumps are five bytes.

```
$ rasm2 -a x86 -b32 "jmp 0x41414141"
e93c414141
```

So what is left to do? Well, we can simply rewind to the first complete opcode five bytes before the `ret`, and add a jump, then relocate the remaining opcodes. In this case, we could do something like this:

```
smashme:
push ebp
mov ebp, esp
sub esp, 0x28
mov eax, dword [ebp + 8]
mov dword [esp + 4], eax
lea eax, [ebp - 0x18]
mov dword [esp], eax
jmp parasite

parasite:
call sym.imp.strcpy
leave
pop ecx
cmp ecx, 0x08048456
jne 0x41414141
jmp ecx
```

Here, `parasite` is mapped someplace else in memory, such as our new section.

With this technique, we'll still to have to pass on protecting a few kinds of function epilogues, such as where a target of a jump is within the last five bytes. Nevertheless, we've covered quite a lot of the intended CFG.

This approach works great on platforms like ARM and MIPS, where all instructions are constant-length. If we're willing to install a signal handler, we can do better on x86 and amd64, but we're approaching a dangerous situation dealing with signals in a generic patching method, so I'll leave you here for now. The code for applying the explained patches is all open source and will soon be extended to use emulation to compute relative calls.

Thanks for reading!
Jeff