Binary protection schemes

by Andrew Griffiths

andrewg@felinemenace.org

Table of Contents

Foreword

The purpose of this document is to show, explain, and break, various protection schemes under Linux. Starting off, we'll have a look at what shouldn't be implemented in a protection system, with practical examples.

The specific protection schemes this document will cover are general licensing type implementations, and modifications to an original binary to protect it or obscure it against analysis.

Due to the nature of this document, previous experience of the reader is necessary. Where applicable, relevant links to material will be made available as footnotes along the document as it progresses.

Along the way, this document will also cover:

- Tricks that you can use compiling binaries
- Using various pieces of software to debug and modify programs
- Licensing scheme implementations
- Binary self-checksumming, obfuscation, and encryption

This document should be both applicable to ELF encryptors, and those implementing commercial programs.

Along the way are exercises for the reader to do if s/he so wishes to. The aim of the exercises is to fortify the readers knowledge by doing the things presented, and identifying the weaknesses in it.

Also, this document will have some pieces of text highlighted so that the readers attention can be drawn to it easily.

Software that may be of interest

If you know of other software that is applicable to this document, feel free to contact the author (see the conclusion section for contact information.) and suggest that it be added.

Free/Open software

- GNU Binutils
  - Used to link, analyse and modify various binary formats.
  - Can be found at: http://sources.redhat.com/binutils/
- GNU Project Debugger
  - Used to debug and analyse programs
  - Can be found at: http://sources.redhat.com/gdb/
  - For the purposes of this document, grab http://www.eccentrix.com/members/mammon/gdb_init.txt (or grab a copy at http://felinemenace.org/~andrewg/gdbinit.txt), a GDB initialisation file that gives gdb some commands similar to Softice. It was written by mammon_, elaine, pusillus and mong. To provide feedback about it, see http://community.reverse-engineering.net/viewforum.php?f=35
- nasm
  - Used to compile assembly code in intel syntax to either a binary representation, or an ELF object file.
  - Can be found at: http://nasm.sourceforge.net
- Fenris
  - A document explaining fenris can be found at http://lcamtuf.coredump.cx/fenris/whatis.shtml
  - To quote from the above url:

    "Fenris is a suite of tools suitable for code analysis, debugging, protocol analysis, reverse engineering, forensics, diagnostics, security audits, vulnerability research and many other purposes. "

  - Can be found at http://lcamtuf.coredump.cx/fenris/
- HT Editor
  - Hex Editor / Disassembler (with cross references etc)
  - Multiple architectures supported.
  - Can be found at: http://hte.sourceforge.net/

- LDasm
  - http://www.feedface.com/projects/ldasm.html
  - Similar to wd32asm for Windows.
- Linux Interactive DisAssembler
  - Can be found at: http://lida.sourceforce.net
  - Does a lot of interesting things.
- The-Dude
  - Can be found at: http://the-dude.sourceforge.net
  - Kernel-level debugger for analysis of binaries.
- Bastard
  - Disassembler environment
  - Can be found at: http://bastard.sourceforge.net
- elfsh
  - Allows you to make assorted changes to ELF files.
  - Can be found at: http://elfsh.segfault.net

Commercial software

- IDA
  - Can be found at http://www.datarescue.com
  - Disassembler for multiple file formats and architectures.
  - Now has a Linux console version. (That isn't LIDA as above, however, in case there was any confusion.)

Skill honing

In order to encourage people to learn more about reverse engineering and protection schemes (specifically under Linux) a PullThePlug[1] game box was set up to help peoples learning.

1 http://www.pulltheplug.org – In general, a like minded community of technically inclined people.

This box has various pieces of analysis software configured on it, and the people running it (the author is one of those people) are open to suggestions on what else is to be put on it.

The general idea for this game box is to put up binaries of various difficultly and allow people to work through those binaries and doing the suggested activities on that binary. People are encouraged to write and send in new binaries (generally along with how you made if it their was any special techniques) and put them up.

To get started with this game box, see http://catalyst.labs.pulltheplug.org

Catalyst complements the other boxes already set-up for people to use and learn on:

- Blackhole – a FreeBSD remote exploitation box, which can be found at http://blackhole.labs.pulltheplug.org
- Vortex – A level based exploitation game, covering multiple areas such as stack overflows, heap overflows, integer overflows, format strings, cryptographic analysis, stenography, binary analysis and reversing random number generators. Vortex runs Linux, and can be found at http://vortex.labs.pulltheplug.org

The PullThePlug network has an IRC and SILC server which people may use. Additionally, there is a mailing list that you can subscribe to, to talk about the games and boxes. For more information, see the PullThePlug website.

Designing protection systems by counter-example

This section is basically a tutorial on breaking weak protection schemes that have been implemented in the past to give the reader an idea on what has been done previously, and methods of doing things to break those protection schemes in order to see how ineffectual they are.

If this section doesn't interest you, skip to the next one.

Why counter-example?

Many of the systems below are classical “text-book” examples of bad protection schemes. The idea is to examine previously implemented protection schemes, work out how they failed, and what “rules” can be derived from this.

To show how easy these systems are broken, this document will cover breaking the applicable pieces of code they construct.

Warming up

The files for this section can be found in counter_example/warming_up.

Throughout the paper there is highlighting on various pieces of text. This is meant to draw the readers eyes towards relevant pieces of information.

When this program starts, it prompts for a password to continue execution. Using the program strings on the binary doesn't show anything immediately obvious.

```
/lib/ld-linux.so.2
_Jv_RegisterClasses
__gmon_start__
libc.so.6
printf
```

```
getpass
strcmp
exit
_IO_stdin_used
__libc_start_main
GLIBC_2.0
PTRh
QVh$
[^_]
Password:
printf
Correct password entered.
getpass
Incorrect password entered.
```

For this exercise, we'll pretend that *ltrace*[2] doesn't exist, and we'll use *gdb*[3] and *objdump*[4] to quickly analyse the binary.

From the quick strings output, we can roughly see that there are several functions used, strcmp, printf, and getpass.

objdump -R check gives the following:

```
check:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049714 R_386_GLOB_DAT    __gmon_start__
08049700 R_386_JUMP_SLOT   strcmp
08049704 R_386_JUMP_SLOT   getpass
08049708 R_386_JUMP_SLOT   __libc_start_main
0804970c R_386_JUMP_SLOT   printf
08049710 R_386_JUMP_SLOT   exit
```

If the below looks weird, its because we're using mammon's gdbinit file. You can get a copy at [5]. You may want to grab a copy before proceeding. If the below output makes no sense to you, at the end of this document is a quick run down of what the various areas mean.

Start GDB on the binary, and place a breakpoint on the getpass function, via **bp getpass**. If GDB complains about the symbol getpass not existing, just type run, Control-C out of the function, and type **bp getpass** again.

2 http://freshmeat.net/redir/ltrace/16567/url_homepage/ltrace.html
3 http://freshmeat.net/redir/gdb/3116/url_homepage/gdb
4 http://freshmeat.net/redir/binutils/754/url_homepage/binutils
5 http://www.eccentrix.com/members/mammon/gdb_init.txt

```
gdb> bp getpass
gdb> run
(no debugging symbols found)...Error in re-setting breakpoint 1:
No symbol table is loaded.  Use the "file" command.
(no debugging symbols found)...(no debugging symbols found)...
________________________________________________________________________________
     eax:00000000 ebx:4015BEDC  ecx:00000001  edx:BFFFFA6C     eflags:00000246
     esi:4015D1B0 edi:BFFFF9F0  esp:BFFFF9BC  ebp:BFFFF9D8     eip:400FC6D0
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t s Z a P c
[007B:BFFFF9BC]---------------------------------------------------------[stack]
BFFFF9EC : 00 00 00 00  DC BE 15 40 - A0 64 01 40  F0 F9 FF BF .......@.d.@....
BFFFF9DC : F8 97 03 40  01 00 00 00 - 64 FA FF BF  6C FA FF BF ...@....d...l...
BFFFF9CC : DC BE 15 40  A0 64 01 40 - 00 85 04 08  64 FA FF BF ...@.d.@....d...
BFFFF9BC : 40 84 04 08  B4 85 04 08 - A3 0A 05 40  D9 96 03 40 @..........@...@
[007B:4015D1B0]---------------------------------------------------------[ data]
4015D1B0 : 6C FA FF BF  00 00 00 00 - 00 00 00 00  00 00 00 00 l...............
4015D1C0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
[0073:400FC6D0]---------------------------------------------------------[ code]
0x400fc6d0 <getpass>:   push   %ebp
0x400fc6d1 <getpass+1>: push   %edi
0x400fc6d2 <getpass+2>: push   %esi
0x400fc6d3 <getpass+3>: push   %ebx
0x400fc6d4 <getpass+4>: sub    $0xa4,%esp
0x400fc6da <getpass+10>:        call   0x400395fd <h_errno+87521>
------------------------------------------------------------------------------

Breakpoint 1, 0x400fc6d0 in getpass () from /lib/tls/libc.so.6
gdb>
```

Since the aim is not to debug or analyse libc, tell GDB to return via **pret**.

```
gdb> pret
Password:
________________________________________________________________________________
     eax:0804A170 ebx:4015BEDC  ecx:0804A000  edx:00000169     eflags:00000286
     esi:4015D1B0 edi:BFFFF9F0  esp:BFFFF9C0  ebp:BFFFF9D8     eip:08048440
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t S z a P c
[007B:BFFFF9C0]---------------------------------------------------------[stack]
BFFFF9F0 : DC BE 15 40  A0 64 01 40 - F0 F9 FF BF  A0 84 04 08 ...@.d.@........
BFFFF9E0 : 01 00 00 00  64 FA FF BF - 6C FA FF BF  00 00 00 00 ....d...l.......
BFFFF9D0 : A0 64 01 40  00 85 04 08 - 64 FA FF BF  F8 97 03 40 .d.@....d......@
BFFFF9C0 : B4 85 04 08  A3 0A 05 40 - D9 96 03 40  DC BE 15 40 .......@...@...@
[007B:4015D1B0]---------------------------------------------------------[ data]
4015D1B0 : 6C FA FF BF  00 00 00 00 - 00 00 00 00  00 00 00 00 l...............
4015D1C0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 B0 06 08 ................
[0073:08048440]---------------------------------------------------------[ code]
0x8048440:      mov    %eax,0xfffffffc(%ebp)
0x8048443:      movl   $0x80485bf,0x4(%esp)
0x804844b:      mov    0xfffffffc(%ebp),%eax
0x804844e:      mov    %eax,(%esp)
0x8048451:      call   0x8048310
0x8048456:      test   %eax,%eax
------------------------------------------------------------------------------
0x08048440 in ?? ()
gdb>
```

The program just returned from where it was called from. The line at *0x08048440* is moving the return code from getpass() to a stack variable, the line after that is loading an address at 4+(esp), and then at lines *0x0804844b* and *0x0804844e*, moving the return value from getpass() to (esp).

The AT&T syntax denotes that you are dereferencing a value when its inside braces. To convert it into pseudo-C, the operation is like:

```
esp[1] = 0x080485bf
esp[0] = eax;
eax = 0x08048310();
```

Since recent GCC compilers do not fix up *%esp* after function calls[6], it is harder to determine the amount of arguments passed to the function. However, for this code, its safe to assume its two parameters.

Investigate the program a little by checking out what is stored in *%eax* and *0x080485bf*.

```
gdb> x/s 0x080485bf
0x80485bf <_IO_stdin_used+15>:   "printf"
gdb> x/s $eax
0x804a170:       "password"
```

Since password was what the author entered when prompted for it, this looks like a string comparison. Since we didn't enter '*printf*' as our input, it is likely that this is the password we need.

To determine what function is going to be used, we'll analyse the code at that location in memory.

```
gdb> x/3i 0x08048310
0x8048310 <_init+40>:   jmp    *0x8049700
0x8048316 <_init+46>:   push   $0x0
0x804831b <_init+51>:   jmp    0x8048300 <_init+24>
gdb> x/x 0x08049700
0x8049700 <_GLOBAL_OFFSET_TABLE_+12>:   0x08048316
```

This piece of code is part of the Procedure Linker Table[7] (PLT), which uses an array of pointers, the Global Offset Table (GOT) to determine where execution should continue.

We can examine what GOT entries refer to what functions, to do this, use

6 From what the author has seen, you can basically reduce it to the esp[array] as shown above.
7 url on plt

objdump -R.

```
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049714 R_386_GLOB_DAT    __gmon_start__
08049700 R_386_JUMP_SLOT   strcmp
08049704 R_386_JUMP_SLOT   getpass
08049708 R_386_JUMP_SLOT   __libc_start_main
0804970c R_386_JUMP_SLOT   printf
08049710 R_386_JUMP_SLOT   exit
```

In this case, we can see that once PLT resolving has taken place, the GOT entry at 0x08049700 will contain a pointer to strcmp, and the string compare operation will take place.

Since we know what function we are calling, usually there is no need to analyse its operations and determine what it is doing.

Put a breakpoint on the *testl* (*0x8048456*) instruction and continue execution.

```
gdb> bp 0x08048456
Breakpoint 2 at 0x8048456
gdb> c
_______________________________________________________________________________
     eax:FFFFFFEF ebx:4015BEDC  ecx:00000072  edx:72016E78     eflags:00000293
     esi:4015D1B0 edi:BFFFF9F0  esp:BFFFF9C0  ebp:BFFFF9D8     eip:08048456
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t S z A p C
[007B:BFFFF9C0]---------------------------------------------------------[stack]
BFFFF9F0 : DC BE 15 40  A0 64 01 40 - F0 F9 FF BF  A0 84 04 08 ...@.d.@........
BFFFF9E0 : 01 00 00 00  64 FA FF BF - 6C FA FF BF  00 00 00 00 ....d...l.......
BFFFF9D0 : A0 64 01 40  70 A1 04 08 - 64 FA FF BF  F8 97 03 40 .d.@p...d......@
BFFFF9C0 : 70 A1 04 08  BF 85 04 08 - D9 96 03 40  DC BE 15 40 p..........@...@
[007B:4015D1B0]---------------------------------------------------------[ data]
4015D1B0 : 6C FA FF BF  00 00 00 00 - 00 00 00 00  00 00 00 00 l...............
4015D1C0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 B0 06 08 ................
[0073:08048456]---------------------------------------------------------[ code]
0x8048456:      test   %eax,%eax
0x8048458:      jne    0x8048468
0x804845a:      movl   $0x80485c6,(%esp)
0x8048461:      call   0x8048340
0x8048466:      jmp    0x8048492
0x8048468:      cmpl   $0x0,0x8(%ebp)
------------------------------------------------------------------------------

Breakpoint 2, 0x08048456 in ?? ()
gdb>
```

We can see from *%eax* being *0xffffffef* that the test instruction isn't going to give us a 0, so the jump to *0x08048468* will be taken.

Since the eax == 0 code will call another function with what appears to be a single parameter, lets check that parameter out.

```
gdb> x/s 0x080485c6
0x80485c6 <_IO_stdin_used+22>:   "Correct password entered.\n"
```

This looks like the code path that we'd like to take. We have three options available:

- Change the execution flow for the current execution
- Patch the program memory
- Patch the program on disk.

Pick your own adventure.

Changing the flow of execution

There are several things we could do change execution at this point. For example, *eax* could be set to zero, or the zero status flag could be toggled, which is what we'll cover here.

To change the flow of execution, enter **stepi** into gdb and look at the screen.

```
gdb> stepi
________________________________________________________________________________
     eax:FFFFFFEF ebx:4015BEDC  ecx:00000072  edx:72016E78     eflags:00000382
     esi:4015D1B0 edi:BFFFF9F0  esp:BFFFF9C0  ebp:BFFFF9D8     eip:08048458
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I T S Z a p c
[007B:BFFFF9C0]---------------------------------------------------------[stack]
BFFFF9F0 : DC BE 15 40  A0 64 01 40 - F0 F9 FF BF  A0 84 04 08 ...@.d.@........
BFFFF9E0 : 01 00 00 00  64 FA FF BF - 6C FA FF BF  00 00 00 00 ....d...l.......
BFFFF9D0 : A0 64 01 40  70 A1 04 08 - 64 FA FF BF  F8 97 03 40 .d.@p...d......@
BFFFF9C0 : 70 A1 04 08  BF 85 04 08 - D9 96 03 40  DC BE 15 40 p..........@...@
[007B:4015D1B0]---------------------------------------------------------[ data]
4015D1B0 : 6C FA FF BF  00 00 00 00 - 00 00 00 00  00 00 00 00 l...............
4015D1C0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 B0 06 08 ................
[0073:08048458]---------------------------------------------------------[ code]
0x8048458:      jne    0x8048468
0x804845a:      movl   $0x80485c6,(%esp)
0x8048461:      call   0x8048340
0x8048466:      jmp    0x8048492
0x8048468:      cmpl   $0x0,0x8(%ebp)
0x804846c:      jne    0x804847a
-------------------------------------------------------------------------------
0x08048458 in ?? ()
gdb>
```

Since the Jump if Not Equal instruction checks the zero flag which is currently unset (lower case, highlighted 'z' above). , the execution flow will change to *0x08048468*. We can change this by using the *cfz* instruction, which toggles the state of the Zero flag[8].

```
gdb> flags
o d I T S z a P c
gdb> cfz
gdb> flags
o d I T S Z a P c
```

Now that's all left to do is continue execution, and enjoy the correct password response.

```
gdb> continue
Correct password entered.

Program exited normally.
```

Patching the program on-disk file and process memory

To tell GDB to write your changes to disk, type **set write on** in GDB first. You'll have to stop debugging the program you are running as well, as the kernel locks the file for writing when it is running.

There are a couple of options here to patch.

For a starters, the Jump if Not Equal instruction could be changed to a Jump if Equal instruction, which would make it that if the correct password is entered, it fails. Looking at the below, we can see that the *jne* instruction is 2 bytes long. (*0x0a – 0x08 = 0x02*).

```
0x8048458:      jne    0x8048468
0x804845a:      movl   $0x80485c6,(%esp)
```

Lets examine what two bytes they are:

```
gdb> x/2c 0x8048458
0x8048458 <main+52>:    0x75    0xe
```

8 http://www.posix.nl/linuxassembly/nasmdochtml/nasmdoca.html – section A.2.2

The second byte looks like an offset to where we want to jump. Since jump / call <relative offset>'s take into account the size of the jump / call instruction, the offset is subtracted by two. We can prove this by adding the two numbers together and seeing where we end up:

```
gdb> x/3i 0x804845a + 0xe
0x8048468 <main+68>:    cmpl   $0x0,0x8(%ebp)
```

Now, lets see about that first byte, *0x75*. Looking at the nasm[9] documentation, we see that:

```
    B.4.128 Jcc: Conditional Branch

Jcc imm                       ; 70+cc rb            [8086]
Jcc NEAR imm                  ; 0F 80+cc rw/rd      [386]
```

Since our base is 0x70, we need to find out what the conditional number we need to add to it to change it to Jump if Equal. The nasm documentation goes on to refer to the conditional codes.

```
   In the following descriptions, the word `either', when applied to two possible
trigger conditions, is used to mean `either or both'. If `either but not
   both' is meant, the phrase `exactly one of' is used.
     * O is 0 (trigger if the overflow flag is set); NO is 1.
     * B, C and NAE are 2 (trigger if the carry flag is set); AE, NB and NC are 3.
     * E and Z are 4 (trigger if the zero flag is set); NE and NZ are 5.
     * BE and NA are 6 (trigger if either of the carry or zero flags is set); A and
NBE are 7.
     * S is 8 (trigger if the sign flag is set); NS is 9.
     * P and PE are 10 (trigger if the parity flag is set); NP and PO are 11.
     * L and NGE are 12 (trigger if exactly one of the sign and overflow flags is
set); GE and NL are 13.
     * LE and NG are 14 (trigger if either the zero flag is set, or exactly one of the
sign and overflow flags is set); G and NLE are 15.

   Note that in all cases, the sense of a condition code may be reversed by changing
the low bit of the numeric representation.
```

We see that by changing the least significant bit of the first byte, changing the meaning of the instruction. Lets change the instruction to Jump if Equal via **set *(unsigned char *)0x08048458 = 0x74**. To change it back, you can use **set *(unsigned char *)0x08048458 = 0x75**.

Alternatively, the two byte jump instruction could be nopped[10] out using **nop**

9 http://nasm.sourceforge.net

10 nopped out means that the bytes for that instruction have been overwritten with No Operation instructions, which means that there is no instruction to execute, or alternatively, effectively does nothing, like exchanging a register with itself.

0x08048458 and **nop 0x08048459**. We could force the conditional code to always be true by changing the *test* instruction to *xor eax, eax*. Looking at the nasm manual again for the *xor* instruction, we see that its defined as:

```
XOR r/m32,reg32               ; o32 31 /r           [386]
```

The *r/m32* refers to that this instruction may either be a memory location, or that it may be a register. Lets see if changing the first byte of the test instruction does what we want, via feeding GDB **set *(unsigned char *)0x08048456 = 0x31**.

```
gdb> set * (unsigned char *)0x08048456 = 0x31
gdb> x/3i 0x08048456
0x8048456 <main+50>:    xor    %eax,%eax
0x8048458 <main+52>:    jne    0x8048468 <main+68>
0x804845a <main+54>:    movl   $0x80485c6,(%esp)
```

Success. Sometimes its not as straight forward as just overwriting a byte. When you're not sure what it requires, write the instruction you want to insert, and assemble it with nasm, and then use ndisasm -b 32 to see the opcodes.

Once you've made your changes, verify them (via *x/10i 0xaddress*), and type 'c' to continue execution.

My preferred way of patching is thusly:

```
gdb> set * (unsigned char *) 0x0804843b = 0xeb
gdb> set * (unsigned char *) 0x0804843c = 0x1d
gdb> set * (unsigned char *) 0x0804843d = 0x90
gdb> set * (unsigned char *) 0x0804843e = 0x90
gdb> set * (unsigned char *) 0x0804843f = 0x90
```

because it removes the call to getpass and strcmp. The nops where added so that the disassembly listing wouldn't break.

0xeb is chosen because it represents a short jump, which has a range of 128 bytes. For more information, see [11], specifically relating to the byte representation of the instruction. To verify what the bytes are for an instruction, you can use the *x/4c* instruction in gdb to print the bytes used.

11 http://www.posix.nl/linuxassembly/nasmdochtml/nasmdoca.html#section-A.88

0x1d was calculated due to

```
0x804843b:      call   0x8048320
0x8048440:      mov    %eax,0xfffffffc(%ebp)
0x8048443:      movl   $0x80485bf,0x4(%esp)
0x804844b:      mov    0xfffffffc(%ebp),%eax
0x804844e:      mov    %eax,(%esp)
0x8048451:      call   0x8048310
0x8048456:      test   %eax,%eax
0x8048458:      jne    0x8048468
0x804845a:      movl   $0x80485c6,(%esp)
```

0x5a (90) – 0x3b (59) = 0x1f (31).

Since the parameter for jumps / calls take into account the existing length of the instruction, we need to subtract two bytes from 0x1f, giving us 0x1d.

Keeping it all in plain sight

The files for this section can be found in counter_example/keeping_it_all_in_plain_sight

The system here uses a shared library (This document will still be here for those of you who would like to reminisce about when software authors would use protection schemes that used single calls to DLL's.)

We'll start by examining the library to see what symbols it exports that can be called:

```
 objdump -T liblibrary.so

liblibrary.so:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
<snip>
00000000      DF *UND*  00000039  GLIBC_2.0    localtime
00000754 g    DF .text  00000068  Base         timetrial_check
00000000      DF *UND*  00000010  GLIBC_2.0    time
00000000      DF *UND*  0000001b  GLIBC_2.0    strlen
000007bc g    DF .text  000000ab  Base         is_valid_serial
<snip>
```

The highlighted section shows where the symbol is from (Base if its in the file itself), and the function name.

This shows the library contains two functions, timetrial_check, and is_valid_serial. (If you don't think this is based on reality, see[12].)

Doing a *objdump -d* on the library and following the output shows that the *timetrial_check* and *is_valid_serial* both return 1 when the conditions are met, otherwise they'll return 0.

From this point, we have several options: patch the calling binary, patch the library, or perform some other trick

Patching the calling binary could be tedious work, especially if they call it at various places. Patching the library would be easier, but that's not as clean as it could be.

Another alternative is to *replace* the entire library with a dummy version. This can be achieved by the following code:

```
int is_valid_serial() { return 1; }
int timetrial_check() { return 1; }
```

This dummy library can be compiled via:

```
gcc -shared -Wl,-soname=liblibrary.so evil.c -o evil.so
```

Lets test this code in action:

```
mv liblibrary.so liblibrary.so.old
mv evil.so liblibrary.so
./binary
This binary is unregistered time trial
```

This looks like a wrap now.

12 http://www.woodmann.com/fravia/project7.htm

Turning a program against itself

The files for this section can be found in counter_example/turning_a_program_against_itself.

Running the program, we see the following:

```
./main: user_name company_details license_number
```

Lets do a quick check to see if the file has any hard coded strings:

```
$ strings main
...
OaKmzqAln813Uqsz
k39!aqkJbZok3Asd
bGerKl4v1z2bIOvA
%s: user_name company_details license_number
binary_protection_schemes
License details incorrect
Thank you for registering
...
```

This seems good. Those strings at the top look a little suspicious, lets see where they are referenced. To do this (and make it different from other approaches so far), we'll use GDB and a read breakpoint on the string. Firstly, though, we'll have to calculate where this string is in memory.

strings will print out the hex address of the file if asked to, and from there, we'll need to work out how this correlates to the virtual memory layout of the program.

```
9c0 OaKmzqAln813Uqsz
9d1 k39!aqkJbZok3Asd
9e2 bGerKl4v1z2bIOvA
a00 %s: user_name company_details license_number
a2e binary_protection_schemes
a48 License details incorrect
a63 Thank you for registering
```

Doing a cursory analysis on this binary via *objdump -fp* reveals the following information:

```
main:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
```

```
start address 0x080483d0

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off    0x00000114 vaddr 0x08048114 paddr 0x08048114 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000a7e memsz 0x00000a7e flags r-x
    LOAD off    0x00000a80 vaddr 0x08049a80 paddr 0x08049a80 align 2**12
         filesz 0x00000118 memsz 0x00000180 flags rw-
 DYNAMIC off    0x00000a90 vaddr 0x08049a90 paddr 0x08049a90 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000128 vaddr 0x08048128 paddr 0x08048128 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

We can calculate our virtual address by:

```
0x08048000 + 0x9c0 = 0x080489c0
```

Lets start GDB and put a read watchpoint on the mysterious first string from above. Watchpoints are implemented with hardware support, if the hardware supports it, or GDB will single step the application and see if its been read or modified at the end of each instruction. In either case, watchpoints don't modify the application code.

```
gdb> rwatch *0x080489c0
Hardware read watchpoint 1: *0x80489c0
gdb> set args andrewg hihi internet++
gdb> r
(no debugging symbols found)...Hardware read watchpoint 1: *0x80489c0
(no debugging symbols found)...Hardware read watchpoint 1: *0x80489c0
(no debugging symbols found)...Hardware read watchpoint 1: *0x80489c0
_______________________________________________________________________________
     eax:6D4B614F ebx:BFFFFA10  ecx:00000004  edx:BFFFFA18     eflags:00000286
     esi:4015D1B0 edi:BFFFF990  esp:BFFFF8C0  ebp:BFFFF958     eip:080484A2
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t S z a P c
[007B:BFFFF8C0]---------------------------------------------------------[stack]
BFFFF8F0 : B8 81 04 08  34 F9 FF BF - C8 6F 01 40  01 00 00 00 ....4....o.@....
BFFFF8E0 : 78 F9 FF BF  CF 83 00 40 - 96 82 04 08  8E FF 77 01 x......@......w.
BFFFF8D0 : D8 7C 01 40  40 6C 01 40 - 14 70 01 40  96 82 04 08 .|.@@l.@.p.@....
BFFFF8C0 : 2C 74 02 40  08 7A 01 40 - 03 00 00 00  A8 7C 01 40 ,t.@.z.@.....|.@
[007B:4015D1B0]---------------------------------------------------------[ data]
4015D1B0 : 18 FA FF BF  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
4015D1C0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
[0073:080484A2]---------------------------------------------------------[ code]
0x80484a2:      mov    %eax,0xffffffd8(%ebp)
0x80484a5:      mov    0x80489c4,%eax
0x80484aa:      mov    %eax,0xffffffdc(%ebp)
0x80484ad:      mov    0x80489c8,%eax
0x80484b2:      mov    %eax,0xffffffe0(%ebp)
0x80484b5:      mov    0x80489cc,%eax

Hardware read watchpoint 1: *0x80489c0

Value = 0x6d4b614f
```

```
0x080484a2 in ?? ()
gdb>
```

Bingo, we appear to of found where it has been referenced. It may appear a little weird at first, but breakpoints are triggered after an instruction has been executed on Intel platforms (this may be the case on other platforms as well). The actual instruction that referenced our memory is at *$eip – 5*. By working backwards through memory via *$eip – x* where x keeps increasing, we determine that that function starts at *0x08048494*.

Browse through the function code a little bit (*x/40i $eip* for example). After quickly scrolling through the code, you see its littered with xor's, binary shifts amongst other things. As opposed to fully reversing this binary (and writing a key gen), or analyse the binary, further, lets check the stack back trace.

```
gdb> bt
#0  0x080484a2 in ?? ()
#1  0x4002742c in ?? () from /lib/tls/libc.so.6
<snip>
#6  0x40016c40 in ?? () from /lib/ld-linux.so.2
#7  0x40017014 in ?? ()
#8  0x08048296 in ?? ()
#9  0xbffff978 in ?? ()
#10 0x400083cf in _dl_lookup_versioned_symbol () from /lib/ld-linux.so.2
#11 0x0804884d in ?? ()
#12 0xbffffb5f in ?? ()
<snip>
```

Nothing too unusual, bar their could be better stack information. There are two entries pointing to the the code segment. Lets check out what instructions those are:

```
gdb> x/20i 0x08048296
0x8048296:      pop    %edi
0x8048297:      pop    %edi
0x8048298:      insb   (%dx),%es:(%edi)
0x8048299:      imul   $0x6174735f,0x63(%edx),%esp
0x80482a0:      jb     0x8048316
0x80482a2:      pop    %edi
0x80482a3:      insl   (%dx),%es:(%edi)
0x80482a4:      popa
0x80482a5:      imul   $0x6c727473,0x0(%esi),%ebp
```

We immediately notice that this doesn't seem right. Either there is self-modifying code running, or it's a lack of frame pointers inside of glibc. We'll apply Occam's [13] razor here.

13 http://pespmc1.vub.ac.be/OCCAMRAZ.html

```
gdb> x/s 0x08048296
0x8048296:       "__libc_start_main"
```

Indeed it is the simpler answer of the two. Moving onto the next possible return address.

```
gdb> x/10i 0x0804884d
0x804884d:      mov    %eax,%edx
0x804884f:      mov    (%ebx),%eax
0x8048851:      mov    %eax,0x4(%esp)
0x8048855:      mov    %edx,(%esp)
0x8048858:      call   0x8048358
0x804885d:      test   %eax,%eax
0x804885f:      je     0x8048879
0x8048861:      movl   $0x8048a48,(%esp)
0x8048868:      call   0x8048398
0x804886d:      movl   $0x1,(%esp)
```

OK, so it looks like we've reached a classical if(compare_function (what_we_supplied, what_they_calculated) != 0) {}test.

Now, how to beat his.. we could patch the binary, but we'll do something different. We'll turn this program into a key generator for us. If we poke around the binary and reconstruct what its doing, (read from *0x804881e* to *0x8048848*), we'll see that the function it calls in the end takes three values, and based on what it is doing, it probably looks like (slightly more readable to reflect what would be coded):

```
if(strcmp(generate_serial(username, company, “binary_protection_schemes”), argv[3]) !=
0) {
        <you are unregistered, good bye>
}
```

Since we know what arguments this function takes, we could write our own assembly code to call this function.

In order to call the serial generation function, we're going to have to push 3 values (“binary-protection-schemes”, company, and username, respectively), and then write the returned value (stored in *eax*) to the user.

Ideally, our code inserted into the program shouldn't have hard coded values, but this isn't necessarily available (due to calls to other functions).

With these restrictions in mind, we can create our code to insert into the program. We'll write it in assembly just for the sake of it (we could use C, but more effort would be involved to make it work correctly). We'll use nasm to compile our code (*nasm -f bin patch.asm*).

```
BITS 32                         ; Tells nasm to generate 32 bit code

_start:
        jmp .forwards
.backwards:
        pop eax                 ; EAX now contains the address of product.
        push eax
        add eax, 26             ; EAX now points to Carribean Shipping
        push eax
        add eax, 19             ; EAX now points to Long John Silver
        push eax

        mov eax, 0x08048494     ; we do an indirect call because we can't be
        call eax                ; bothered calculating the offset needed.
                                ; This value was obtained above when looking
                                ; where one of the strings where located.

        xchg ecx, eax           ; ECX now contains the serial number
        xor ebx, ebx
        mul ebx                 ; eax and edx = 0 now
        mov al, 4               ; value needed for sys_write (see
                                ; /usr/include/asm/unistd.h for more information.
        inc bl                  ; ebx = 1 = stdout
        mov dl, 64              ; We'll write 64 bytes of output (key is smaller
                                ; than this, but anyways.
        int 0x80                ; Call the kernel

        mov al, 1               ; EAX = 1
        int 0x80                ; exit out of this process.

.forwards:
        call .backwards


product db 'binary_protection_schemes', 0x00
company db 'Caribbean Shipping', 0x00
person db 'Long John Silver', 0x00
```

Most patches can avoid being so 'brutal' on the application via modifying the call statement to point to their own function, which then will call the serial number generation function.

Now that we have our patch we wish to insert into the program, where should we write it? The most obvious one that comes to mind is the entry point, since it appears that the program doesn't dynamically create any stuff needed to generate the serial number.

Now, to calculate where the file offset is (scroll up if you don't remember how to get that value (Start address under objdump.))

```
(Value of what we want – Load address) + file_offset = absolute_file_address

0x080483d0 – 0x08048000 = 0x3d0
0x3d0 + 0 = 0x3d0
```

We can apply our patch to the program with the dd, a program to copy data between files. The command to apply the patch to the file is:

```
cp main main.patched
dd bs=1 seek=976 if=patch of=main.patched conv=notrunc
```

Running main.patched provides:

```
s8Ul1O1K-akaq!kks-lrGbbbbb
```

And testing our original main with “Long John Silver” “Caribbean Shipping” and 's8Ul1O1K-33qJd9!s-lrGbbbbb' provides:

```
Thank you for registering
```

Indeed, this binary has became its own worst enemy.

Conclusion

What can we learn from the above “protection” schemes that have been implemented in the past?

Do not:

- make any (useful) comparisons with expected codes or generated code, as this will lead to binary patching.
  - From this, we notice we shouldn't use a validation routine, but write code inline to the function that wants to check. This makes more work for people who wish to crack the software.
  - Additionally, this means we can write “fake” code with various strings that don't get called.co
- give meaningful names to your protection code
- use system calls for checking license time. It's better to check the date on multiple files you open, with the added benefit of this being surreptitious since programs often stat() a file before opening them, or fstat()ing them afterwards.
- leave debugging information in the end binaries shipped to customers.
  - Why help people who want to analyse your software?
- put the license checking functions close together.
- Don't use existing libraries for some operations. If you rewrite some of them, it means there is more work for the attacker to do.
- have a single point of failure for your licensing schemes.

So, what can we do?

- Create multiple code paths for your code, based upon input information (such as serial number data). This means that it has became a lot harder to find the path to the “correct” places. If changes are done along the way, the more effort that needs to be expended.
- Checksum your code from multiple places doing different operations based on the checksum results.
  - If you do detect a modification, it would be a bad idea to:
    - Tell the user they caused it to happen. Calling your customers pirates and/or other negative things won't do well for your business (indeed,

because the files could have trivially been modified by a virus for example).

Additionally, telling the user straight away that the software has been modified will give the attacker knowledge that there is some form of self checking or consistency checking happening. If you leave it for a couple of days, it lets the attacker think they've successfully broken it.

- Ensure all the appropriate sanity checking is taking place. For example, reverse the is_valid_serial for the section entitled “Keeping it all in plain sight”, and see if you can spot the biggest flaw in that algorithm.
- try and obfuscate your code and strings as much as possible.
- attempt to separate cause and effect for the results.

Methods for implementing license schemes

Aims

The aims of a licensing scheme is balanced between several things:

- Simplicity

  You don't want to burden or annoy your customers by the protections and what they have to do to use the software they have already paid for.

- Strength against:
  - Analysis

    The program and what's its doing should be heavily protected against analysis by other people.

  - Brute force

    You don't want your license information to be “quickly” brute forced to a correct, usable key. A key which passes the first test but makes your program not that useful, is fine however.

- Implementation

  It should be feasible to implement in your program, and should make sense to implement it.

- Provability of showing who released their license information or codes.
- Deterrence against people giving it out

To “correctly” implement a (decent) licensing scheme for your product will take time, effort, and research.

There is no magical wand you can wave to make your products secure against cracking. Using pre-made software for protecting binaries / files is generally disliked, due to if a cracker reverses one program protected with a pre-made software, it will most likely help said cracker to reverse other applications that use the same pre-made code to protect binaries.

There is definitely a lot of stuff to consider when embarking to write such a protection/license system.

Discussion

Here are some points for thought when embarking upon your mission of creating a protection / license system. After working out what your aims are, and what trade-offs you are making, now you need to consider how it will interact with other things.

- Is your methods used for the program going to annoy your legitimate customers?
  - Will you have to increase your own resources to handle customer queries?
  - What will be your turn around time for handling them?

- Does implementing your scheme make sense for your product?
  - Are you going to spend too much time preventing people from copying your product, when the time could be better spent improving your existing software?

- Is the appropriate code confusing enough to deter some crackers[14]?
  - Do you copy your license code information around in subtle ways?
    - Making the license information harder to track will help frustrate analysis of the implementation.
  - Are their redundant copies?
    - Having multiple copies of the data in memory makes it harder to search for the one being used in the program.
  - Is it used to make logic choices in the other parts of the program?
    - The more the correct execution of the program relies on the correct number, the harder it is to patch.
  - Is the code spread out throughout the program as opposed to being

14 As has been said on various sites I've visited when looking for random things, a true cracker will relish this challenge.

    localised?

    - Having the code spread out through the program makes it slightly harder to analyse the binary, especially when you're not sure what you're looking for.

- Have you turned on optimisations on for your code?
    - Compiler optimisations can make it trickier to understand what a program is doing.

Complex number of checks

A particular method that can be used quite effectively to slow down[15] crackers attacking a piece of software is to have "configure" the software based upon the serial information / license information.

An implementation of this method can be described as follows:

1. Use the input value to configure a random number generator.
2. Randomly[16] seed a set of structures defined like:

```
struct reg_nodes {
        int state;                 // Is this value on or off?
        int linked[LINKED_NUM];    // What other structures am I linked to?
        int link_num;              // How many of these links do i have
} reg_nodes[NODE_NUM];
```

    1. Randomly set the state to either 0 or 1 for all the reg_nodes[x].state
    2. Define a random number of links to other structures.
    3. Insert the appropriate values into the linked values.
3. Pick a reg_node between 0 and NODE_NUM – 1 based on the serial / license / key information, and toggle its state(0 -> 1, 1 -> 0)
4. Traverse the linked entries on the chosen structure, and toggle their state. (NOTE: don't parse the children's children.)
5. Change the operations of the current node.
    1. You could add another node to the children list if you have space.

15 It has been said several times the key to not having your software cracked is often just boring the cracker to death.
16 You'll need a PRNG which can give you the same output based on a seed value.

    2. You could reduce the amount of nodes it is connected to if the node has any connections.
    3. You could randomly swap two nodes if the node have two or more child nodes.
    4. You could randomly replace a child link if the node has any.
6. If there is more license / serial information, goto 3 .
7. Use calculated data in your program. You could use it as logic choices, options to pass to functions, etc.

The aim is to provide a method that complex to calculate the output trivially based on input, and given only the output states for each node, difficult/impossible to work it into what was fed into the algorithm.

Here are some ideas that can be used to use the information that has been generated above:

- Function arguments

  We can use parts of the data generated from the license key to make parameters that will be passed to other functions. An example might be:

```
/*
•  O_RDONLY is defined as 00 on Linux.
•  O_WRONLY is defined as 01 '       '.
•  O_RDWR is defined 02 '        '.
*/

int save_file(...) {
        /*
         *The save_file function won't operate as expected if reg_nodes[0].state
         *does not contain 1.
         */
        fd = open(path, O_CREAT|O_TRUNC|reg_nodes[0].state, 0700);
}

int read_file(...) {
        /*The read_file function won't operate correctly if reg_nodes[1].state is 1.*/
        fd = open(path, reg_nodes[1].state);
}
```

  If you need more bits set for argument types, you can bit-shift them together to make a value.

  For example, if you wanted to combine the first 16 reg_nodes[] entries to make a short int, you could use the following type of code:

```
combined = 0;
for(i=0; i < 16; i++) combined |= (reg_nodes[i].state << i);
```

This now gives you a 16 bit value that you can use for operations. If you wanted to combine your licensing scheme with how you obscure your binary from analysis, you could use the technique in “Indirect code flow change” in order to make it so that without the key, it becomes a lot harder to follow the correct program flow.

Another area where this license-data generated data could be put to use in initialising certain pieces of data in your program. For example, if it was a printer driver, you could use this to initialise the colouring values (inside appropriately complex data structures, of course). Without the correct value there, the output would be distorted or rubbish, or whatever effect you'd like.

- Logic Choices

The generated license code could be used throughout the program to make logic decisions that may have a impact on the program through slightly (to extreme) faulty logic.

When implementing code that makes use of the logic choices, there is a general rule to aspire to: When a logic choice is made, its results should be subtle, and they should manifest themselves further in the future, in order to hinder locating where the logic choice was made.

Because the results happen later in the program, it means that there is a greater amount of code that an attacker must examine to find out where it happens. The more subtle the cause of the result, the greater chance they will miss it.

This in turn makes it a lot harder for an attacker to analyse said programs.

A simple example of this choice would be:

```
if(reg_nodes[2].state != 1) {
       /* correct code here */
} else {
       /* slightly buggy code here */
}
```

Making things slightly more difficult:

```
if(reg_nodes[3].state == 0) {
       /* incorrect code here */
       /* other random checks */
} else {
       /* correct code here */
       if(reg_codes[5].state != 0) {
              /* correct */
       } else {
              /* slightly break results */
       }
}
```

The main point here is to use your imagination and find places to put the code where you would find difficult to analyse (for example, complex functions).

- "Property"/Data choices

  By discreetly using the license data as property modifiers, you can make it so that while it appears something may be working fine, the end result isn't what they expected.

  A good example perhaps might that if you were writing a game, it could be used to decide:

  - what levels are accessible (if the game is sequential, not being able to play level 4 would be annoying.
  - if certain power ups / modifiers are effective
  - the "luck" of the player
  - whether or not certain end bosses are killable

In general, to attack this method, you would have to identify all[17] checks in the

17 If you are aiming to enable a certain subset of functionality, you may be able to get away with less.

program, and determine[18] what state they are in to enable / allow certain operations to happen, and/or what there effects are.

The above is only for proof of concept; while decent, you shouldn't directly access the values created, instead, copy them around the place to many places, and then access them once they've been moved around lots (and incorporated into, and removed from, and thoroughly copied everywhere in your program) as this will make the time spent for the cracker attacking the software somewhat exponential.

If you are going to use this type of system, you'll need to clearly state to your customers that the software will only work **correctly** when the **correct** serial / license information has been entered.

If you wanted to reduce the likelihood of a typographical error, you could insert a trivial check byte or checksum (using, perhaps, CRC-16) so that you could inform the user that they entered the key incorrectly. Doing this additional checksum also provides a slight diversion to an attacker as well, as they may think this is all that is needed to crack the piece of software.

Since the security of this algorithm is based upon the number of bits you use, and how they are used, you would probably want to generate a new sequence of which bits must be set a certain way for the correct operation of your program each major release.

To generate license keys for your customers using this method is slightly tricky, there are a couple of methods available:

- Brute force the randomiser and string until you reach a valid string for them. This may not sound like much, but the worst case for *n* bits is O(2^*n*). For example, if you where using 16 bits, it would be 65,536 operations, and for 24 bits, it would be 16,777,216 operations, and for 32 bits it would be 4,294,967,296 operations to verify the correctness of the key.
- Use a genetic algorithm[19] to search for a key. Depending on the key space, and complexity of the algorithm you use to calculate what reg_nodes should

18 This could be done by patching the program and restarting, or disassembling and analysing those codes protected by the check and seeing if the cracker can determine which is the appropriate code path to follow.
19 http://gaul.sourceforge.net – an implementation of a GA library in C.

be, this may just provide “close-enough”'s but not any hits.

- Write a key generator that keeps state of all the reg_nodes, and analyses what input would better move it towards a complete key for that user. This would have to find which input would bring the number of bits to what they are meant to be so the program can be used, and the least amount of bits that would change the state of existing, currently good, bits.

This protection method (based on randomisation) could be extended in various ways. For example, creating a chain (or, alternatively, a graph) of function pointers which control the program flow which in turn would guide program execution. If the incorrect function pointers are there, the program results are erratic / going to crash.

Server contact

The files for this section can be found in licensing/server_contact

The general method behind this approach is to have the software contact a server who can verify the request / give the client applicable code or data to continue execution. As you've probably gathered by now, it is not sufficient to do *if (I_am_allowed_to_continue_executing() == 1) { /* do stuff */ } else { /* complain */ }*

One correct way to implement such a check is to get a bunch of data used for program operation, so that without this information the program won't work (possibly correctly, depending on the nature of the program). The server could make a decision to give the client what it needs based on its serial number amongst other pieces of information (if needed to correctly enforce the licensing scheme.)

Another method of doing this would be to obtain various functions (depending on multiple conditions, you may be able to send the actual binary to the client) so that it can continue execution.

If the binary is going to be contacting an external server under your control, and the person has to type in some authentication material, you can use a system like Secure Remote Passwords (SRP[20]) or One Time Passwords (OTP) to prevent someone from key logging / monitoring what password they entered on the machine. Using SRP or OTP is only really useful when the person using the software may end up using it on a box that isn't under their control, and you'd like to avoid analysis of the binary.

This approach would be most effective against certain forensic analysis tools[21], and would fit in well for a robust ELF protector.

In case of ELF protectors, SRP or OTP isn't going to be useful if they calculate their response to the server on the machine that the binaries connection is originating from. It would be the only option in this case for them to calculate it on another machine, if you want to prevent people recording the information required to generate the result that causes the binary to run.

For commercial programs, one big thing you'll want to keep in mind is your customers. What happens to the programs they've already paid for in case your business ceases to exist? What happens if your server isn't answering requests at the moment?

Additionally, this method may discard your program from being accepted in various business areas because they will not allow your program to bypass the firewall, especially for trivial things such as licensing, unless they **absolutely** require your program. Another area is where individuals think your program is invading their privacy. So while this method is one of the more resistant methods to attack, there are many considerations to take into account before implementing it.

Encrypted functions / data

A particularly effective measure for shareware[22] authors that distribute full programs, but restrict functionality is to encrypt certain parts of their program they don't want the unregistered people to have, is to have the license number build a suitable decryption key, or alternatively, a license file using asymmetrical encryption (such as RSA or ECC)

20 http://srp.stanford.edu
21 http://www.honeynet.org/tools/sebek/
22 I use the term shareware as a distribution method.

To get you started thinking about this, here is a sample implementation using license keys.

Use 12 random bytes, use the first four bytes to seed a random number generator, and the rest is used to cycle through a pseudo random number generator number, and extract a byte, and append it to a string. Once it reaches 8 bytes, use that to decrypt the data, with a suitable cryptographic algorithm.

To make the 12 random bytes easy for the customer to type in, you can encode it to base64, which will make the key length 16 bytes.

Another method to generate a decryption key might be to use a finite automaton (see [23] for more information about finite automata and how it could be applied). This example isn't the best in the world, but it is sufficient and applicable to the task at hand:

```
unsigned char keyspace[400];
int key_upto;

unsigned char *crypto_block[] = {
        “\xde\xad\xbe\xef”,
        NULL,
        “\xfe\xe1\xde\xad”,
        < ... from 0 -> 31 entries>
};

int next_state(int current, char c)
{
        // This is effectively a compressed lookup table
        //
        //   | 0 1 2 3 4 5 ...
        // --|----------------
        // 0 | 0 1 2 3 4 5 ...
        // 1 | 1 2 3 4 5 6 ...
        // 2 | 2 3 4 5 6 7 ...
        // . | . . . . . . ...
        // etc.
        //
        // Whenever state 1 is reached (crypto_block[1]), the loop below terminates.
        // If more effort was put into it, you could do use the input to solve
        // a polynomial and use the result as a return value.

        return (current + c) % 32;
}

void parse_key(char *license_key)
{
        state = 0;

        // make sure we don't overflow keyspace..
```

23 http://www.ics.uci.edu/~eppstein/161/960222.html

```
          while(crypto_block[state]) {
                  state = next_state(state, *license_key);
                  if(crypto_block[state]) {
                          memcpy(keyspace + key_upto. crypto_block[state], 4);
                          key_upto += 4;
                          license_key++;
                  }
          }
  }
```

The benefits of doing this is that its difficult to determine what the decryption keys are, what length they are, and if the next_state function is complex enough, what states produce what output.

The benefits of finite automata is that:

- it becomes extremely difficult to work out what is happening without the data used to run it and obtain a correct result, as there is no test for correctness. Especially when it it used to configure things, with mostly incorrect paths, but enough "paths" for the amount of license keys you need.
- There aren't many / any conditional checks, so there is nothing to patch.
- There is more work involved to model what your program is doing. (More so on the attackers side, as s/he doesn't have the benefit of documentation)

Finite Automata can be extended and used in other ways. A more involved document applying a similar concept can be found at [24].

Another approach that could be taken is that if the potential customers are downloading the software off your website make you can make *n* versions of your software, and generate *n* keys (and perhaps a certain amount of different decryption keys, and use *#ifdef's* to control which code block is used), and encrypt each version with a different key. For each download, choose a different encrypted version. This will help reduce the effectiveness of key generators written by people who don't have all the copies that exist.

While this method does not stop someone cracking it after registration (you can only make it take longer to do, nothing with current technology can be crack-proof it appears), it will slow down/stop the people who haven't paid for the software, assuming its implemented properly.

24 http://www.codebreakers-journal.com/viewarticle.php?id=33&layout=abstract

Conclusion

This section has documented several resistant to attack methods that could be used inside an application that meet several of the aims above.

As we've seen in the above examples, for the protection scheme to improve the resistance against attack, the protection scheme must be a crucial part of the operations of the binary.

With this section done, lets move onto the binary modifications section, but before that, I have a question to the readers:

If you published how your license scheme/algorithm worked, does this help break the implementation of it? Compare and contrast with encryption algorithms and their descriptions / publicly available implementations.

Binary modifications

Introduction

The aim of modification of the binaries is to increase its resistance to attacks, such as reverse engineering, debugging, patching and analysis by other people.

This section will mainly cover the ELF format as used by most recent UNIX-like systems such as Linux and FreeBSD amongst others, however, this section will be equally applicable to other formats.

If you need a quick refresher on ELF binaries, have a look at the appendix entitled 'A brief overview on ELF'. If that doesn't answer your question, the ELF specification should.

We'll start with some of the more simple modifications you can do to the binaries, as the difficultly of understanding and being able to implement such as system (or series of systems) rises.

Aims

There are several things we want to aim towards when thinking about modifying binaries:

- What are we trying to achieve, exactly?
  - Do the proposed methods achieve this?
- Will the methods used reduce the portability of the program?
- Does it have an impact on performance?
  - Is it localised at startup time, or spread across execution of it?
- Does these modifications have certain side effects?

  For example, if you do self decompressing executables, the system won't have a cache of the file in memory when next executed, thus leading to a (albeit, slightly) longer start up time.

- Depending on the nature of the executable, whether or not distributing it would be a good idea might come into play. if you absolutely don't want it analysed, don't distribute it. (It may seem obvious..)

Encryption

Obfuscation of the .text segment

The files for this section can be found in binary_modifications/obfuscation.

As a warm-up, this method shows a relatively trivial method of just hiding the binary code of an application.

Taking a look at the general program headers for */bin/ls* gives:

```
/bin/ls:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08049a50

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x00000100 memsz 0x00000100 flags r-x
  INTERP off    0x00000134 vaddr 0x08048134 paddr 0x08048134 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00011d08 memsz 0x00011d08 flags r-x
    LOAD off    0x00012000 vaddr 0x0805a000 paddr 0x0805a000 align 2**12
         filesz 0x000003f4 memsz 0x000007b0 flags rw-
 DYNAMIC off    0x00012184 vaddr 0x0805a184 paddr 0x0805a184 align 2**2
         filesz 0x000000d8 memsz 0x000000d8 flags rw-
    NOTE off    0x00000148 vaddr 0x08048148 paddr 0x08048148 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
EH_FRAME off    0x00011cdc vaddr 0x08059cdc paddr 0x08059cdc align 2**2
         filesz 0x0000002c memsz 0x0000002c flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
```

From this, we note what the start address is, and file offsets. Since we now have everything we need to obfuscate this binary, lets make a start.

Firstly in order to encrypt the binary, we'll need somewhere to store our code, both in the file, and so its mapped into the program memory. We could use the page-size padding technique to modify our file, but since we want to encrypt things separately, we'll nuke a useless program header (such as NOTE), and write our new program header there, and append our code to the file.

We'll need a virtual address to map our code to, in this example we'll use 0x50000000.

The NOTE program header refers to what type of system the binary was from (and compilers amongst other things.) Since its not important in most cases, we can just overwrite this header to make a new piece of code is mapped into our program..

For reference, the *elf.h* currently has the below values defined.

```
/* Solaris entries in the note section have this name.  */
#define ELF_NOTE_SOLARIS        "SUNW Solaris"

/* Note entries for GNU systems have this name.  */
#define ELF_NOTE_GNU            "GNU"
```

Our “protection” for this binary will consist of a trivial xor operation, since this is just to get people warmed up for following, and doing, the other ones.

Secondly, we'll need a decryption loop to put inside the program that will decrypt the text segment of the binary (often, but not always, starting at 0x08048000, and ending at the next program header in the binary)

A suitable decryption loop is shown below.

```
BITS 32
ORG 0x50000000

_start:
        pusha                           ; The below values get patched later
        mov ebp, 0xdeadbeef             ; Pointer to entry_point
        mov ecx, 0xfee1dead             ; Length of decryption loop

        mov esi, ebp
        add esi, ecx                    ; Terminating address.

        xor edx, edx                    ; EDX is used to decide which byte value
                                        ; we are going to or from.

decrypt_loop:
        mov al, byte [keydata + edx]    ; Pick a predefined byte to use for
                                        ; xoring.
        xor [ebp], al
        inc ebp
        inc edx

        cmp edx, 8
        jnz over

        xor edx, edx

over:
        test ebp, esi                    ; have we reached the end?
        loopnz decrypt_loop              ; nope, keep going.

        popa
        mov eax, 0x11223344              ; Fixed up with e_entry in the ELF header.
        jmp eax                          ; change execution to entry_point


keydata db 0xca, 0xfe, 0xba, 0xbe, 0xb0, 0x0b, 0x1e, 0x50
```

Now that we have this all the prerequisites done, we have to:

- Parse the ELF header and program headers and find:
  - Entry point of program (from this we get: length of data we need to encrypt, where to start encrypting data from in the file)
  - Whether or not we have a NOTE program header to overwrite.
  - File offset and virtual address for first LOAD program header.
- Read in the appropriate data from the file
- Encrypt the program data
- Modify the NOTE program header so it is a LOAD program header, and the appropriate entries pointing to our virtual address and our file offset. To keep it simple, we will pad the file with 0's so our address lies on a page boundary.
- Modify the first LOAD program header to make it a writeable segment so that a segmentation violation isn't caused when we try to write to that area.
- Write the file back out.

The file encrypt.c is an implementation of what can be done to add an obfuscation layer to the program.

The general (and immediate) method to attack this protection is to wait until the program has reached the original code, and to “dump” the decrypted values, and overwrite the encrypted values in the program with the original.

Some exercises for the reader:

- extend that program and make it encrypt parts of the .data segment (stuff not needed for the binary to load.)
- Ask for a password and use that to decrypt the binary (via a trivial method, and then a “proper” scheme using say sha-1, rc4 or aes-128).
- Think about methods of combining it with license info.
- Write an unpacker for this scheme. Can you think of some generic methods?

Loading executables in user-space

The files for this section can be found in binary_modifications/decompression.

As opposed to what the UPX[25] team says about self-decompressing not being possible under Linux, it is, and you'll be able to do it after reading this section.

To achieve this, we need to 1) not clash on address space with the binary we're decompressing, and 2) load any libraries needed by the program.

To avoid the address space clash with the program who we'd like to load, we can tell the linker to compile our program to use a different address space (not 0x0804xxxx). To modify the system linker script to suit our purposes, we'll modify the default script.

To get the default linker script, run *ld -verbose > linker_script* and remove the lines up to and including the first === and every line (including itself) after the next ===. When we compile the decompression program, we'll have to specify a custom linker script to gdb via *-Wl,--script=linker_script* for it to output binaries with what we want in them. After that, modify the script and replace references to 0x08048400 to 0x00001000 (for example).

Examine the binary with objdump after compiling it with the *-Wl,--script* option and see what you got.

25 http://upx.sourceforge.net

```
example:     file format elf32-i386
architecture: i386, flags 0x00000 112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00001290

Program Header:
    PHDR off    0x00000034 vaddr 0x00001034 paddr 0x00001034 align 2**2
         filesz 0x000000e0 memsz 0x000000e0 flags r-x
  INTERP off    0x00000114 vaddr 0x00001114 paddr 0x00001114 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x00001000 paddr 0x00001000 align 2**12
         filesz 0x00000484 memsz 0x00000484 flags r-x
```

This now means that are program will start execution at 0x00001290 and won't clash for 0x0804xxxx range.

Now we need to extract the relevant information from the binary so we can load it into memory.

In order to keep things simple, we are going to use dietlibc[26] in these examples. This allows us to avoid having to load libraries needed for the programs.

Our dietlibc test binary has the following program headers defined:

```
dietlibc:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080480a0

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00001904 memsz 0x00001904 flags r-x
    LOAD off    0x00001904 vaddr 0x0804a904 paddr 0x0804a904 align 2**12
         filesz 0x0000005c memsz 0x00000068 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

To load this binary successfully, we need to map it into memory.

We need to load from offset 0 in the file to 0x08048000 for 6404 (0x1904) bytes, and then load from offset 6404 to 0x0804a904.. wait a second, that isn't correct.

To load a binary into memory is slightly more complex than that, with various rules that apply (page alignments for various things).

26 http://www.fefe.de/dietlibc/

The algorithm needed to correctly load a program header entry into memory looks like:

```
for(i = 0; i < program_header_num; i++)
        check_if_its_a_loadable_segment();
        slop = virtual_address & (page_size – 1)
        base = virtual_address - slop
        mmap(base, program_header_file_size + slop, protection_flags, MAP_flags..,
                file_descriptor, file_offset – slop)
}
```

Read over *loader.c*.

Now since the file is mapped into memory, we need to transfer control to do it. Firstly, however, we need to either create an initial stack layout, or use the original one. Since the former is more reliable (and I already have code available to do, so it doesn't require much effort to include here) we'll use that.

Read over *main.c*. For a greater understanding of what's needed to load binaries inside userspace, see [27]

create_stack() takes two options, firstly, the amount of arguments the next one has, and an array of char pointers used for argv. It builds a stack layout like:

```
<null>
<terminating null for envp>
<terminating null for auxv entries>
<argv[n]>
<argv[n-1]>
...
<argv[0]>
<argc>
```

When you are loading the file in line however, it would most likely be more reliable (and correct) to reuse the existing stack for the application[28], and saves having to allocate spurious bytes.

Running *./loader dietlibc* provides the following:

27 http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-01/0002.html
28 Depends how you implement the loader. If its done in C and has the start up lib, the stack is modified to call main, which means you would have to modify it if its going to call the start up lib code in the other binary.

```
        [+] Elf loader opening dietlibc
                [+] 3 program headers
                [+] Program header 0 is of type 1:1
                [+] Program header 1 is of type 1:1
                [+] Program header 2 is of type 1685382481:6474e551
                        [-] Skipping non-load segment
Calcuating stack information
        Stack address is 0x20003ffc
        Need 18 bytes to store the argv strings
        Need 72 bytes to store argc, argv, envp, auxv info
        This leaves 16290 bytes left for the program for the running program
Jumping into program
I am a diet libc compiled program whose execution flow was originally in another
binary.
00001000-00003000 r-xp 00000000 03:02 377898     /.../decompress/loader
00003000-00004000 rw-p 00001000 03:02 377898     /.../decompress/loader
00004000-00025000 rw-p 00004000 00:00 0
08048000-0804a000 rw-p 00000000 03:02 316342     /.../decompress/dietlibc
0804a000-0804b000 rw-p 00001000 03:02 316342     /.../decompress/dietlibc
20000000-20004000 rw-p 20000000 00:00 0
40000000-40016000 r-xp 00000000 03:02 13846      /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 03:02 13846      /lib/ld-2.3.2.so
40017000-40019000 rw-p 40017000 00:00 0
40024000-40154000 r-xp 00000000 03:02 13871      /lib/tls/libc-2.3.2.so
40154000-4015d000 rw-p 0012f000 03:02 13871      /lib/tls/libc-2.3.2.so
4015d000-40160000 rw-p 4015d000 00:00 0
bffff000-c0000000 rw-p bffff000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

The output is from displaying the parents memory maps *(cat /proc/pid/maps*), which clearly shows the two binaries existing in the same process space. Based on this, we can say that it is possible to implement binary compression under Linux.

The output looks a little weird because the code is from other projects of the authors.

Exercises for the reader:

- Extract the appropriate information from the (or use the whole) binary, and implement compression and modify the loader to use memory as opposed to reading from files.
- Load the appropriate libraries needed for the program to execute.
- Remove *all* traces of the other libraries and memory mappings before letting the application properly execute (ie, more than, say, 3 instructions for the target executable)
- Modify the stack creator so that it will take into account environmental variables.

Tying binaries to a host

It may be advantageous to (help) restrict execution of binaries to a single host[29]. This can be done by taking fingerprints of the host system and using the data provided by them in the operation of the program (for example, it could decrypt the next stage of the program).

There are various pieces of information on a system that you can use to restrict it.

For example:

- Machine host name
- Machine ethernet MAC addresses
- /proc/partitions
- /proc/pci
- /proc/version
- /proc/net/route
- CPU serial number if available
- Network card MAC address(es)
- etc

can all be used to help restrict binaries to certain hosts. In case your customers' machines change a small amount, you could use a form of error correction codes (such as reed-soloman) to reconstruct the data. This would allow you to limit the amount of changes that are allowed while still working.

Obviously, as you should be able to tell by now, its not going to be sufficient by checking the data is the same each boot up. To do it properly, you'd have to use the data in another way, like an encryption layer.

The availability of this information also allows you to do other things. Consider, for example, that you wanted to have long term public key / private key encryption in a host, but you can't store information easily, or trust

29 http://www.packetstormsecurity.org/groups/teso/indexsize.html – burneye. This has an option to do this.

worthily.

On each program start up, you could generate several identifying pieces of information (in case that a single piece, or multiple pieces, change), and several decryption keys, because the information will stay relatively the same for a long period of time. This allows you to encrypt a piece of information, and send the encrypted data, and your public identifying host-hash across the network, and retrieve it next program start up.

This could perhaps be implemented via doing a cryptographically strong hash across the host information to generate the public key, and then reversing the host information to make the encryption key.

Per page encryption

The files for this section can be found in binary_modifications/perpage.

Per page[30] encryption means to encrypt each feasible page with a unique key. To make the data get decrypted, the code needs to know what and when certain pieces of data is accessed. One method of implementing this is to mark the pages with no read, write or execute privileges. Once that data is referenced, it generates a segmentation violation (SIGSEGV). If we “hook” SIGSEGV via *sigaction()*, we can get the faulting address that was referenced, decrypt the page, remember the page so we know to re-encrypt it later on, and then return from the signal handler to continue execution of the program as if nothing had happened.

The *sigaction()* handler also lets us do many types of interesting things, specifically inspecting the program to why it crashed.

The sigaction handler is defined as:

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

The second argument is siginfo_t, which is defined as

30 http://jamesthornton.com/redhat/linux/8.0/System-Administration-Primer/s1-memory-virt-details.html

```
siginfo_t {
    int      si_signo;  /* Signal number */
    int      si_errno;  /* An errno value */
    int      si_code;   /* Signal code */
    pid_t    si_pid;    /* Sending process ID */
    uid_t    si_uid;    /* Real user ID of sending process */
    int      si_status; /* Exit value or signal */
    clock_t  si_utime;  /* User time consumed */
    clock_t  si_stime;  /* System time consumed */
    sigval_t si_value;  /* Signal value */
    int      si_int;    /* POSIX.1b signal */
    void *   si_ptr;    /* POSIX.1b signal */
    void *   si_addr;   /* Memory location which caused fault */
    int      si_band;   /* Band event */
    int      si_fd;     /* File descriptor */
}
```

The si_errno and si_code can be used to determine what caused the segfault so you can take appropriate action. The address that caused the problem is stored at si_addr.

The 3rd argument is the ucontext_t one, which is defined as (see /*usr/include/sys/ucontext.h* for more information):

```
/* Type for general register.  */
typedef int greg_t;

/* Number of general registers.  */
#define NGREG   19

/* Container for all general registers.  */
typedef greg_t gregset_t[NGREG];

/* Context to describe whole processor state.  */
typedef struct
  {
    gregset_t gregs;
    /* Due to Linux's history we have to use a pointer here.  The SysV/i386
       ABI requires a struct with the values.  */
    fpregset_t fpregs;
    unsigned long int oldmask;
    unsigned long int cr2;
  } mcontext_t;

/* Userlevel context.  */
typedef struct ucontext
  {
    unsigned long int uc_flags;
    struct ucontext *uc_link;
    stack_t uc_stack;
    mcontext_t uc_mcontext;
    __sigset_t uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
  } ucontext_t;
```

The “gregs” contain general purpose registers which we can interrogate, and if we like, modify them so that when the signal handler is returned, certain things are modified, such as EIP, or the value of EAX, etc.

The implementation provided along with this paper does a slightly modified loading method from the previous section. After loading the file in via mmap, it generates a random encryption key and creates a structure.

The encryption used is a simple sliding xor, as the point of this is to show examples of how things can be done, not to provide a ready to roll code base.

Running the supplied code which loads an elf file, encrypts the pages, marks them as untouchable, and then decrypts them as needed provides the following output.

```
        [+] Elf loader opening dietlibc
                [+] 3 program headers
                [+] Program header 0 is of type 1:1
page 0..
page 1..
                [+] Program header 1 is of type 1:1
page 0..
                [+] Program header 2 is of type 1685382481:6474e551
                        [-] Skipping non-load segment
Calcuating stack information
        Stack address is 0x20003ffc
        Need 18 bytes to store the argv strings
        Need 72 bytes to store argc, argv, envp, auxv info
        This leaves 16290 bytes left for the program for the running program
Jumping into program
SIGSEGV handler
siginfo: 0x20003c30, ucontext: 0x20003cb0
 - Faulting address: 0x80480a0
 - si_errno: 0
 - si_code: 2
 - EIP of process: 0x80480a0
 - Found segv_lookup entry, making page readable and decrypting
 - Done, continuing execution
SIGSEGV handler
siginfo: 0x20003c28, ucontext: 0x20003ca8
 - Faulting address: 0x804ac08
 - si_errno: 0
 - si_code: 2
 - EIP of process: 0x80480ab
 - Found segv_lookup entry, making page readable and decrypting
 - Done, continuing execution
SIGSEGV handler
siginfo: 0x20003b28, ucontext: 0x20003ba8
 - Faulting address: 0x80498e0
 - si_errno: 0
 - si_code: 2
 - EIP of process: 0x80483ad
 - Found segv_lookup entry, making page readable and decrypting
 - Done, continuing execution
I am a diet libc compiled program whose execution flow was originally in another
binary.
00001000-00003000 r-xp 00000000 03:02 307813     /.../perpage/loader
00003000-00004000 rw-p 00001000 03:02 307813     /.../perpage/loader
00004000-00028000 rw-p 00004000 00:00 0
08048000-0804a000 rw-p 00000000 03:02 307777     /.../perpage/dietlibc
0804a000-0804b000 rw-p 00001000 03:02 307777     /.../perpage/dietlibc
20000000-20004000 rw-p 20000000 00:00 0
40000000-40016000 r-xp 00000000 03:02 13846      /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 03:02 13846      /lib/ld-2.3.2.so
40017000-40019000 rw-p 40017000 00:00 0
40024000-40154000 r-xp 00000000 03:02 13871      /lib/tls/libc-2.3.2.so
40154000-4015d000 rw-p 0012f000 03:02 13871      /lib/tls/libc-2.3.2.so
4015d000-40160000 rw-p 4015d000 00:00 0
bffff000-c0000000 rw-p bffff000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

We can extend this method to re-encrypt pages so that we have the least amount of visible data applicable. This would be implemented by storing the pages that have been decrypted, and then encrypting them next time the handler is executed.

More often that not, binaries will utilise the heap for storing (possibly sensitive / stuff you wouldn't want disclosed straight away) information in. The heap usage can be tracked by using the *brk()* system call when the program starts up and recording that information.

Later on in program execution, you'd do a *brk()* call and store the result, and later on in the program do another *brk()*, to see if the heap size has increased. If it has increased, you can encrypt the page contents and change the permissions on that page so access to that page will cause a segfault (allowing you to decrypt the applicable page.)

This method allows you to encrypt as much as possible for the program, however, it has a variable impact on your programs performance (depends how you implement it, and allow how many “hot” pages you allow to be open.)

Depending how this method is implemented, it may be attacked by writing a custom program using *ptrace()* to attach the program and loop over each page and attempt to single step or continue execution, which in turn will raise the appropriate signal and probably decrypt the page.

To help prevent attacks against this method, you can map out how pages are related[31] to each other and utilise that information in working out how to decrypt the page.

Per function encryption

The files for this section can be found in binary_modifications/perfunc.

Per function encryption works by encrypting each function with a unique key, and modifying the function data to call the appropriate decryption function (which will encrypt the previously called function), and then possibly modify the stack layout to make the function return decrypt the previous function, and re-encrypt the function that the it had just left.

There are two main methods you can use to implement this technique.

- Via signals (and keeping a record of the first byte)
- Via calling a decryption routine, and keeping a record of the bytes overwritten.

To implement the signals method of doing this, find a signal handler, and an opcode that will generate that signal. When that signal is generated, analyse the information given to the signal handler to work out where it was, and decrypt.

To implement the calling method, keep a record of the bytes overwritten, and then encrypt the function start address + bytes_overwritten. The overwritten bytes should then be replaced with a *save_state; call decryption_routine*. Because the application you are interjecting your code into will be using the registers, and flags, you need to make a copy of them. (Via *pusha; pushf* and other registers you need to keep a copy of. Using *pusha* and *pushf* would make the overwritten byte count 7.)

Out of the two methods shown, signal handlers have a slightly better advantage that it can be called by one or two bytes you need to put in the function as opposed to the "huge" amounts of bytes that would be overwritten with push instructions and calls, which can be fingerprinted. The slight advantage

31 Be wary of passing data to libc, as that may cause unexpected challenges if you are going to use this method.

however disappears when you run the program interactively, as then you can place the appropriate breakpoints in the program image.

To encrypt each function requires locating the function start, and how long it is. We can use the programs symbol table to achieve this.

In this example, we will encrypt a handful of functions, *first*, *second*, *third*, and *fourth*. In a proper implementation you would want to parse the symbols, and then encrypt as many functions as possible. Some default included functions don't tend to lend themselves well to being encrypted, as they don't include their length in the symbol table.

If you know a function can only be called from a single or several places, you can use that to restrict where the function is called from. The general aim of this would be to help frustrate binary analysis by making it harder to analyse that particular function without knowing where is was called from. An implementation of this idea would be, for example, use the calling EIP address as part / used in the decryption process.

This method has a variable impact on a programs performance, but is somewhat resistant to analysis. If the program was to take code flow information into account, it could detect attempts to try and get it to decrypt arbitrary functions, and take suitable action, such as corrupting the program image so that other areas that might be dumped won't properly work.

The included implementation of this idea utilises signals to decrypt functions as needed, and modifies the return address to point to *0xdeadbeef*, and uses the SIGSEGV signal to change execution back to the parent function.

Exercises for the reader:

- Make the code encrypt the parent function, and encrypt the called function once its finished executing.
- Implement the pushf/pusha/call method.

Conditional code obfuscation

The files for this section can be found in binary_modifications/conditional.

The general idea behind conditional code obfuscation[32] is to implement program flow logic outside of the programs binary, and move it elsewhere. This is achieved by removing all jumps and calls (conditional or not) and replacing the opcodes with something else that generates an appropriate signal to allow the signal handlers to take control of the program, and determine what should be done next.

The included code used to demonstrate this uses libdisasm[33] to analyse the binary to identify jump codes. The ICEBP instruction is used because the faulting address (si_addr in the info parameter of the sigaction signal handler) is the instruction that caused it, however, there would be more opcodes that could be used. Some ideas of signals to look at would be SIGILL, SIGFPE, and SIGBUS.

To compile the code, run *make* first, and then run the analyse program over the included test program and then compile the loader for this (*gcc -Ilibdisasm_src-0.21-pre1 -Llibdisasm_src-0.21-pre1 loader -Wl,--script=.linker_script loader.c -o loader*), then run *./loader test* to see it work.

```
Calcuating stack information
        Stack address is 0x20003ffc
        Need 14 bytes to store the argv strings
        Need 72 bytes to store argc, argv, envp, auxv info
        This leaves 16294 bytes left for the program for the running program
Preparing to jump into cyberspace... hold on.
Inside sigtrap_handler
--> si_signo: 5
--> si_errno: 0
--> si_code: 1
--> faulting address: 0x80480bb
--> Found entry
--> EFLAGS: odItsZaPc
   --> Got a conditional eip moving thingy :)
   --> Is the zero flag cleared?
   --> Nope
   --> Conditionals wasn't met, moving EIP to next instruction
```

As you can see, there is some similar code between this and the last one. The instruction scanning included in analyse.c is misguided because it would be more efficient to scan each function (as obtained from the symbol table), as opposed to tracking executable flow.

~~jumps/codes before.~~ If there is prior work let me know so I can attribute / mention it here.
33 http://bastard.sourceforge.net

However, this code is included because it was written before the author realised it was a misguided method of analysing the program, and the other code that does function scanning is tightly integrated with another project of the authors.

```
08048115 <otherstuff>:
 8048115:       55                      push   %ebp
 8048116:       89 e5                   mov    %esp,%ebp
 8048118:       83 ec 04                sub    $0x4,%esp
 804811b:       c7 45 fc c8 00 00 00    movl   $0xc8,0xfffffffc(%ebp)
 8048122:       83 7d fc 00             cmpl   $0x0,0xfffffffc(%ebp)
 8048126:       f1                      icebp
 8048127:       90                      nop
 8048128:       f1                      icebp
 8048129:       90                      nop
 804812a:       ff 4d fc                decl   0xfffffffc(%ebp)
 804812d:       f1                      icebp
 804812e:       90                      nop
 804812f:       c9                      leave
 8048130:       c3                      ret
```

As you can see from the above, it becomes more difficult to analyse the binary without any call / jump instructions. For a greater impact on performance, more things could be emulated, such as *leave* and *ret* instructions. Since this code uses a signal/single (play on words) byte instruction to achieve what it needs, *nops* are used to pad the instructions. Random bytes could be used instead, which will possibly help frustrate analysis of the binary, by breaking the disassembly of the program.

The included method for determining what logic to take and apply is straightforward, and doesn't really add all that amount of time to reversing it.

This method has the nice advantage that there is no longer any more jump / call instructions in the binary. This implies the attacker has to reconstruct the instructions in the binary so that they can then analyse the binary properly. Before they can do this though, they must determine how the choices are made, and you can make even more work for them before they even get to this point in time.[34] A method of making it harder to analyse would be to use a virtual machine (see “Virtual CPU” later on in the document) to control the results.

A general attack against the signal handler method would be to modify the stack the signal handler will use so that the various pieces of data like EIP and

34 They may still be able to infer certain things about the program's behaviour, however.

EFLAGS is in a protected page, where as its local variables will be on a non-protected page.

Exercises for the reader:

- Reverse and remove the ICEBP instructions in the included test program.
- Modify the code analyser to do each (practical) function via the symbols available.

Running line

Included for your reading pleasure in binary_modifications/running_line, is a very basic running line[35] proof of concept.

There are several possible uses for running line code, such as:

- Obfuscation

  Decrypting the next instruction that is going to be executed, and re-encrypting the previous one (if the block of code is executed multiple times). This makes analysis slightly more tricky.

- Slowing down debugging.

  This works because for each signal the program will be stopped, and the debugger will take control. Even if its set up to automatically tell the debugger to pass the signal back to the program, their will be a noticeable delay in execution time.

- Self debugging / patching

  If the signal handler keeps state (and, updates state) this could be used to debug or patch parts of your application, such as inserting a jump, toggling the state of the zero flag, etc.

- State

  If you keep and update state you can apply some various tricks, such as changing signal handlers, knowing when to activate tracing inside an application, when to disable it. If the state updating mechanism is complex enough, it will make the attackers job harder to determine what causes signal handlers to change, tracing to be activated, etc.

- Encryption keys

  If you know the code you're about to execute, you could use this method to build a encryption/decryption key.

Implementing running line code is generally tricky to do it correctly.

This technique also has the advantage with per page encryption, as you can restrict access to data pages to a single instruction.

Additionally, correctly emulating running line code is somewhat difficult (full-blown emulators should be able to do it, plus the human controlling a debugger has to be aware what instructions he does which could change the trap flag status), so its possible to use this method to detect if your code is being actively debugged, and possibly emulated.

Another document describing some advanced tricks you can do with running line can be found at [36].

Obfuscation

What is obfuscation?

Obfuscation is the art of obscuring how something does what it does, usually by making it a lot harder to analyse.

Source level

One of the most famous source code obfuscation places can be found at [37],

36 http://www.codebreakers-journal.com/viewarticle.php?id=21&layout=abstract
37 http://www.ioccc.org/

which has competitions on who can write the most obfuscated code possible. However, this doesn't necessarily translate into harder to understand assembly code.

There is document explaining (tongue in cheek) about writing unmaintainable programs[38], which you may be able to get some ideas from, and you may be able to apply it to generated assembly output.

There are some techniques that can be applied is to have different equality checks. For example, if you have a value which can be in two states, you could check it with, for example, *if(value ==0)* or *if(value !=1)* to produce different assembly output. The compiler actually decides how its implemented on the assembly level is up to it, and could differ from what you expect.

Some other approaches is to have code that is interleaved with other code in a program. For example, the processing of a serial number could be included and interleaved in different critical areas, like screen updating, general input handling and so on. When the code is buried amongst other areas, it becomes a lot harder to analyse than a single function call.

Assembly level

It is possible (and feasible[39]) to obfuscate[40] what parts of the binary is doing on an assembly level by including “junk” instructions, obfuscated control flow via opaque jumps, changing the control “blocks” around. For more information, see the footnote 40.

Two-processes

A method than can be used to obfuscate what programs are doing is to have two processes running, and have them swap roles occasionally (the child process becomes the debugger, the debugger the child process). The point of this is to help frustrate analysis. There are some elf encryptors which use this technique such as Shiva[41]. Implemented and used correctly, it can greatly

38 http://mindprod.com/unmain.html
39 http://www.packetstormsecurity.org/groups/teso/indexsize.html - objobf
40 http://www.mysz.org/papers/obfuscation.pdf
41 http://www.securereality.com.au

enhance resistance to runtime analysis of a binary.

Summary

This section has covered many different techniques that can be applied in order to make analysis of a binary harder, and raise the skill level needed to successfully analyse a binary.

In order to keep things simple in this section, it has shown how to do these things with the information in an array. There are other methods of keeping encryption keys and other pieces of information secret. For more information, see the “General things for consideration” section.

One thing to note about modifying the binary, is that, more so in the future, modifications to the binary will have to be done in a way that doesn't make it “obvious” that it has additional code in it, as it will most likely become more common for people to check the expected layout of the binary. (For example, 2 LOAD segments, EIP not pointing into the last page of a load header, etc.) There are other approaches that could be taken to hijack execution, such as modifying the entry point code that ends up calling main indirectly, or disassembling a piece of code to find a call, and modifying it.

Anti-analysis techniques

Runtime analysis refers to analysing the binary by running (or emulating the binary), where as static analysis refers to taking a dead listing[42] and working from that.

Run-time analysis

Emulators

The key to attacking emulators is to work out what they are capable of, and not capable of, and using those things which they can't do to make them die, or alternatively, provide false output. The more work they have to do to bring their emulator up to scratch, the more effective your measures are.

For example, if an emulator you're looking at defeating doesn't support MMX, you would obviously (I hope by now) include some long routines using a fair amount of MMX instructions as opposed to executing CPUID and checking bit 23[43] of EDX to see if it supports MMX and bailing if it doesn't.

In this case, adding a new complete, fully working[44] instruction set to an emulator is a fair amount of effort that an emulator developer would have to exert to support it.

Analysing the general design of a emulator can be fruitful as well. You may be able to find various methods to cause the emulator to run slower than usual.

An example assembly snippet is:

```
BITS 32

_start:
```

42 Dead listing is where you run a disassembler over the code, and analyse the resulting disassembly output, and not interact with the program in a debugger.
43 NASM documentation, B.4.34, CPUID.
44 Indeed, if you can find some quirkities that you can use in the instruction set, all the better, because this means more work of verifying it matches a real machine as much as possible.

```
        xor ecx, ecx
        dec ecx
_me:    loop _me
```

Certain emulators will deal with these types of things better than others.

Another approach for slowing down an emulator would be to use various conjectures, such as Collatz, before continuing the program flow.

Briefly put, Collatz conjecture is a function which, if the input its odd, multiples the input by three, and adds one. If its even, it divides by two, and loops until it reaches one. Collatz conjecture says that for any given input, it will reach one (Eventually.). An example Collatz function is:

```
while(input != 1) {
        if(input & 1) {
                /* odd */
                input *= 3;
                input++;
        } else {
                /* even */
                input /= 2;
        }
}
```

This could be extended as well, by keeping count of how many times the code looped until input became one, and using that value to initialise various things.

Debuggers

Generally, attacking debuggers means to find some general flaws with how it implements things, or general flaws with the debugging interface.

For example, the ptrace() interface is somewhat limited with what you can do with it. If your program implements threads, it means the tool writer must do more work to correctly implement things. Several tools out there don't currently implement support for threads, so implementing threads that do useful things is a good anti-analysis technique.

As well as the limitations of the tools of the person attacking your binary, you can do several things to help slow them down.

For example, a standard anti-debugging technique is to attempt to ptrace() your parent and then detach from debugging it. If you run strace or attempt to gdb that program, the two programs will become deadlocked (both programs are attempting to debug each other). If there is no debugging going on (or the programs are running as different user ids), everything will be fine. The below snippet of code shows what is necessary to achieve this:

```
ptrace(PTRACE_ATTACH, getppid(), NULL, NULL);
ptrace(PTRACE_DETACH, getppid(), NULL, NULL); // note, there is no error checking.
```

A common method in use today to detect debuggers is to use the SIGTRAP handler, and insert *int3* instructions in side the binary. Here is a piece of code that implements this idea:

```
#include <signal.h>

int count = 0;

void int3(int signo)
{
        count++;
}

int main()
{
        signal(SIGTRAP, int3);
        __asm__("int3;");
        printf("The count is %d\n", count);
}
```

Here are the results when the program is run in different methods:

Command	***Result***
./sig	The count is 1
ltrace ./sig 2>/dev/null	The count is 0
strace ./sig 2>/dev/null	The count is 0

As you can see, this method can be used against some tools successfully, other tools will alert you to the presence of such a trick, such as fenris (fenris ./sig) below:

```
...
***************************************************************
* WARNING: I detected a debugger trap planted in the code at  *
* address 0x080483f3. This int3 call is "connected" to a    *
* SIGTRAP handler at 0x080483c4. Please use Aegir or nc-aegir *
* carefully remove this trap, see the documentation.          *
***************************************************************
...
The count is 0
...
```

As Fenris traces through the code, it can (attempt to) detect these things. To find out more about Fenris, see [45].

Fenris did display that the count was 0, but gives you a warning that the binary is playing foul. However, you can use different instructions that call the signal handler, such as *ICEBP*.

Running Fenris with the *ICEBP* instruction (replace the *int3* above with a *.byte 0xf1*) now shows:

```
fenris ./sig
...
The count is 0
...

(no mention that there was a debug trap placed in the binary. However, after lcamtuf
reads over this document, or someone reports it, the author is sure he'll rectify this
issue.)
```

Running the binary using *INT3* as the breakpoint instruction with *gdb* shows:

```
gdb> run

Program received signal SIGTRAP, Trace/breakpoint trap.
0x080483f4 in main ()
```

A human can continue the execution of the program via telling the kernel to pass the signal to the application to handle.

```
gdb> signal SIGTRAP
The count is 1

Program exited with code 017.
gdb>
```

The same could be achieved in Fenris' interactive debugger (Aegir, or nc-

45 http://lcamtuf.coredump.cx/fenris/whatis.shtml

aegir).

Another example of a weakness is that there is an interesting flaw in the kernel ptrace() implementation that Fenris needs to work around. In order to be able to examine signal handler code when the signal handler is called, a break point needs be written to the signal handler address passed to the system call. This can, obviously, be abused to modify instructions, break checksum calculations, or modify data. Fenris, however, does attempt to alert when the signal handler points at certain opcodes that it doesn't expect.

Finding and defanging these things are somewhat easy enough to find however, and the operating system kernel can be modified to make what the program tried to do appear successful, but not actually carry it out, if applicable to the anti-debugging method used.

As well as the obvious ptrace() interface, there are several other methods that can be used to see if you are being debugged. For example, there is */proc/pid/status*, specifically the *TracerPid* line.

Usually people/programs “trace” code by single stepping through all the instructions. Single stepping can somewhat sneakily detected by using the following assembly snippet:

```
pushf
pop eax
and eax, 0x100
```

If eax is not zero, the program is being single stepped, and you can take an applicable approach. As opposed to exiting, or doing other stuff, separate cause and effect and aspire to subtleness. For example, you could slightly modify the licensing schemes (a lot later on in the code) so they won't work, or you could flip a bit which corresponds to a somewhat important, but won't be called for a while, function.

This particular method can be attacked easily by having the tracer look at what instruction is about to execute, and if it is, modify the resultant value on the stack and remove the trace flag.

There are various ways of breaking / detecting a debugger, mostly its a matter of sitting down and analysing how they work and what things they do.

Static analysis

Indirect code flow change

Modifying the code flow via indirect jumps (when done non-obviously) helps frustrate a person looking over a dead listing as it becomes harder to analyse what is happening with a program.

There are several methods available, but the better methods involve other pieces of data/code, the more its done runtime the better. For example, using part of the license key information would be a good method.

This could be achieved perhaps using the following:

```
int part_of_license_key_information;

__asm__("redirect_func:;\n"
        "pushl (%esp);\n"
        "movw part_of_license_key_information, %eax;"
        "xorw %ax, (%esp);"
        "ret;\n");

int main()
{
        part_of_license_key_information = 0xdead;
        redirect_func();
}
```

Doing things similar to this means that there is no determinable place the code will be going to without analysing and having the correct license key information available.

There are some other tricks that can be used to trick (depending on how its implemented) code flow analysis, for example:

```
__asm__(“im_jumping_here:;\n”
       “popl %eax;\n”              ; gets rid of the return value
       “...”);

int main()
```

```
{
        im_jumping_here();

        /* fake code that will never be executed */
}
```

Below are some more traditional methods of breaking disassembly flow.

Inserting bytes in-between instructions

While it doesn't necessarily actually jump into an instruction, it inserts fake bytes into the program to attempt to confuse disassemblers and users by making them think so.

An example might look like:

```
BITS 32

_start:
        jmp .next
db      0xe8
.next:
        xor eax, eax
        inc eax
        int 0x80                     ; exit(ebx)
db      0xff, 0x41, 0x31, 0xe0
```

When compiled into an ELF file, the resulting *objdump* disassembly looks like:

```
08048080 <.text>:
 8048080:       e9 01 00 00 00          jmp    0x8048086
 8048085:       e8 31 c0 40 cd          call   0xd54540bb
 804808a:       80 ff 41                cmp    $0x41,%bh
 804808d:       31 e0                   xor    %esp,%eax
```

This problem is prevalent when the disassembler disassembles blocks of code, and doesn't follow instruction flow.

Under IDA Pro, it looks like:

```
.text:08048080 start:
.text:08048080                 jmp loc_8048086
.text:08048080 ;
.text:08048085                 db 0E8h
.text:08048086 ;
.text:08048086 loc_8048086:                            ; CODE XREF: .text:start
.text:08048086                 xor     eax, eax
```

```
.text:08048088                inc     eax
.text:08048089                int     80h             ; LINUX - sys_exit
.text:0804808B                inc     dword ptr [ecx+31h]
.text:0804808B ;
.text:0804808E                db 0E0h
.text:0804808E _text          ends
```

As can be seen, IDA noticed the the trick byte was placed inside the code, and disassembled the binary appropriately. The last instruction in that disassembly is to be expected, since it appears IDA doesn't terminate disassembly after running into a exit() system call, and there was instruction bytes afterwards.

Use the same bytes for multiple instructions

If the disassembler comes across two ways of being able to display an instruction, it must, obviously, make a choice on how to display this instruction. An example of this would be:

```
BITS 32
_start:
db      0xeb, 0xff, 0xe0     ; effectively jmp _start + 1; jmp eax
```

IDA 4.7 (Linux) has decided to display the representation this way:

```
.text:08048080                public start
.text:08048080 start:                                 ; CODE XREF: .text:start^j
.text:08048080                jmp     short near ptr start+1
.text:08048080 ;
.text:08048082                db 0E0h
.text:08048082 _text          ends
```

These instructions could be displayed several ways, such as the first byte being a *0xeb* byte representation, and then showing a *jump eax* instruction.

Some disassemblers will also choose arbitrary methods of displaying instructions based upon a first come, first choice of representation as well, which can be used to confuse the user. An example of such a disassembler would be the *ht* disassembler.

Dynamic content

The general idea behind dynamic content is to fetch certain/random information of the Internet. An example might be to connect to www.google.com and do a query on something, and then use the 3rd character of the second paragraph for configuring the program.

By fetching information dynamically, it becomes harder to work out what the program is trying to achieve while doing static analysis.

Applicable to both

Opaque conditionals

In order to make analysis of binaries harder via analysing call graphs, you can insert various things which will always be false or true and insert references to global data, calling functions and other things you would find confusing in the section that won't be called.

A trivial[46] example of an opaque jump would be:

```
xor eax, eax
jz somewhere_else
random_instructions
```

This technique works well for both static analysis and runtime analysis if the approach you use for determining the results is complex enough.

Another method of implementing this is to implement threads that both calculate and store to a global variable, and let them race against each other, and the last thread to update the variable is used for the code logic below.

Utilising threads is useful as it means more things for people to keep track of (and thus, greater chance of confusing them).

46 In the sense it would be feasible to automatically find these.

Build code on the stack to execute

The general idea behind this is to build executable code on the stack and jump to the stack to continue execution. From there you can do call other functions or what ever you need to do.

By doing this, you make it harder for static analysis tools to follow what you're doing because they have to implement stack operations and follow calls to it. It helps frustrate runtime analysis because breakpoints on the stack are one time use, which in turn makes single stepping and tracing a pain.

You can extend this method further by putting a canary in the return address, and removing it after you've called a function on the stack. The general approach with this could look like (thanks to raven for the idea):

```
push 'dwords'
call esp
; get random value and store in eax
xor [esp], eax
xor [fn+end], eax

; modify the targets return epilogue to do the following:
xor [esp + ret], dword
```

The general problem with implementing schemes like this are that they drastically reduce portability of your program. For example, there are various kernel patches [47],[48],[49] that won't let you execute code on the stack. Some other operating systems have these features as well, probably notably Windows XP Service Pack 2 for the most part for readers.

Somewhat fortunately they usually allow your program to set various flags to say that it needs to execute code on the stack or heap.

Modifying the ELF headers

Various tools can be made to behave unexpectedly if ELF headers have been modified in ways that can cause confusion in the tool. This section will show

47 http://www.openwall.com
48 http://pax.grsecurity.net
49 http://people.redhat.com/mingo/exec-shield/

various methods of tricking/misleading certain tools, however, it is not meant to be an exhaustive listing of all the tricks available, but hopefully gets people interested in the ELF format, and finding their own tricks to implement.

LIBBFD

Firstly, the classical LIBBFD deserves a mention as it is used by lots of GNU tools. LIBBFD is a library that generically allowing displaying and modification of many types of binary formats. It allows you to examine and modify file formats such as ELF, COFF, PE, SREC, TRAD-CORE amongst others.

A simple example of breaking LIBBFD and some other tools is to set the section header offset size to something silly like *0xdeadbeef*. In order to make these changes, you can use a program called ht[50], or alternatively, write your own tool to make these modifications.

When the section header offset is modified, and attempted to be loaded under gdb[51], the below message is shown:

```
not in executable format: File truncated
```

Of course, when you come across binaries like these, its trivial enough to whip up a correction tool to write over those entries with 0's.

Section table inconsistencies

The files for this section can be found in anti_analysis/section_tables.

The section table is meant to be a more verbose method of describing an ELF file layout. Some of the things that the section table shows are where various areas (such as *.text* and *.bss* are mapped, where a function is located and loaded, and how long the function is.

50 http://hte.sourceforge.net – This is listed in the software section.
51 GNU gdb 6.1-debian was tested.

The section table can be inconsistent with the program headers because for loading the binary the section table is not used at all, which allows us to modify or construct a section table in order to attempt to trick programs or humans.

To get started with this section, we'll take *good_bad.c*, compile it, and then modify the binary using *ht* to modify where the function starts and stops.

Using *ht*, we see that *main()* starts at 0x08048398, and is *0x1e* bytes long. We also see that the harmless() function starts at *0x08048384*, and is *0x14* bytes long. If we swap those values around, and start the binary is IDA, we see that it starts off by displaying the harmless() function (now called main()).

```
.text:08048384                public main
.text:08048384 main           proc near
.text:08048384
.text:08048384 var_8          = dword ptr -8
.text:08048384
.text:08048384                push    ebp
.text:08048385                mov     ebp, esp
.text:08048387                sub     esp, 8
.text:0804838A                mov     [esp+8+var_8], offset aHarmless ; "harmless\n"
.text:08048391                call    _printf
.text:08048396                leave
.text:08048397                retn
.text:08048397 main           endp
```

This function doesn't seem to do much.. However, if we follow the entry point code, we see:

```
.text:080482C0                public _start
.text:080482C0 _start         proc near
<snip>
.text:080482C8                push    eax
.text:080482C9                push    esp
.text:080482CA                push    edx
.text:080482CB                push    offset __libc_csu_fini
.text:080482D0                push    offset __libc_csu_init
.text:080482D5                push    ecx
.text:080482D6                push    esi
.text:080482D7                push    offset harmless     ; This is where main() is.
.text:080482DC                call    ___libc_start_main
.text:080482E1                hlt
```

Which determines that the proper *main()* function is now named *harmless()*.

If we look at the harmless() function, we find that the disassembly for it is different.

```
.text:08048398                 public harmless
.text:08048398 harmless        proc near               ; DATA XREF: _start+17^Xo
.text:08048398
.text:08048398 var_8           = dword ptr -8
.text:08048398
.text:08048398                 push    ebp
.text:08048399                 mov     ebp, esp
.text:0804839B                 sub     esp, 8
.text:0804839E                 and     esp, 0FFFFFFF0h
.text:080483A1                 mov     eax, 0
.text:080483A6                 sub     esp, eax
.text:080483A8                 mov     [esp+8+var_8], offset aMalicious ;
"malicious\n"
.text:080483AF                 call    _printf
.text:080483B4                 leave
.text:080483B5                 retn
.text:080483B5 harmless        endp
```

In order to “break” the disassembly output, all that is needed is to mess around with the *.text* segment and turn change its type from SHT_PROGBITS to SHT_NULL, and IDA PRO won't load that section.

There are more devious things that can be done, such as creating your own section table and symbol table and so on, and provide a completely fake binary representation. The advantages of doing this is that it will trick programs who use the section table. For example, in GDB, you can display fake disassembly listings, etc.

Fake program headers

The files for this section can be found in anti_analysis/program_headers.

Another neat trick to mislead people using IDA PRO analysing binaries that have had the symbol tables completely removed is to add some fake PT_LOAD headers that IDA will load, but the operating system won't load, due to inconsistencies between IDA's ELF loader, and the operating systems ELF loader.

Currently (at IDA 4.7), the IDA ELF loader will load program headers that have a load offset size and virtual address that's modulo to the page size (4096 bytes for this example) is not equal.

```
...
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000004c9 memsz 0x000004c9 flags r-x
    LOAD off    0x000004cc vaddr 0x080494cc paddr 0x080494cc align 2**12
```

```
         filesz 0x00000104 memsz 0x00000108 flags rw-
...
     LOAD off    0x00002fff vaddr 0x08048000 paddr 0x08048000 align 2**2
          filesz 0x000004ca memsz 0x000004ca flags rwx
```

In order for that bottom LOAD header to be loaded by the operating system, the offset size would have to modulo pagesize == 0.

The binary for this section prints one thing, while the disassembly suggests it should be printing another thing.

Final words

This section isn't meant to disparage any particular given tool, it is meant to show that you can't necessarily trust what your tools are telling you, and that sometimes you may have to dig into the headers in order to determine if there is no trickery happening, especially if its a malicious or unknown binary.

If you can afford[52] a copy of Interactive Disassembler (IDA), it is well worth the money if you intend on doing a lot of serious binary analysis, as it handles many different file formats quite well, has a powerful scripting language, plug-in architecture, and has a debugger which may be of use.

There is now a Linux console version available now, which makes it even more worthwhile to the author at least.

If you can't afford a copy, there is a free limited Windows evaluation copy at [53]. And if that doesn't suit your needs, there are many other disassemblers of varying functionality that can be found (see the software section at the top of the document.)

Running line

As previously noted in Modifying Binaries, running line code is extremely efficient method of helping to frustrate analysis of programs, especially in a debugger when all the commands the program is executing keeps on changing.

52 Or, alternatively, get your work to purchase the needed number of copies.
53 http://www.datarescue.be/downloaddemo.htm

Embedded languages

In order to make analysing a binary more difficult, you can embed various scripting languages (such as, Python, Perl, TCL, etc) inside your program, and use that language for some operations. Of course, to be effective, the language you want to use would have to allow itself to be compiled into byte form to make it so that the user doesn't get the textual representation of the code used in the program.

While this isn't a necessarily a strong method of hiding some code logic from prying eyes, it certainly adds more work they have to do because:

- most (all?) disassemblers don't allow the displaying of two different assembly languages at the same time.
- Most disassemblers won't know about the byte code used for the various scripting languages out there, and thus effectively forcing the attacker to examine the byte-code interpreter for the scripting language, and/or use another tool to do work, and slowing them down a bit.
- The attacker may not be familiar with how that scripting language byte-code works, thus meaning they have to do some research to continue reversing that section of the program.

To summarise the above, it could be an effective mechanism to help hinder people analysing how the code works, but how resistant it is will decline as more people get used to having to analyse various scripting languages byte-code representation.

Anti-dumping techniques

“Dumping” a program refers to making a copy of a programs address space, and reconstructing an executable from it. This technique is often used on protection schemes that involve compressing a program without making other changes to the binary. The result of “dumping” the program is that they now have a standard executable to work with (perhaps exactly the same as the binary before the protection was applied to it!), and making reversing the program easier.

The key to thwarting such attempts is to make a dependence inside the binary on other layers of protection outside of it. This section will document and discuss several approaches that can be used.

Dependence on stack and library data segments

Depending on how the dumping tool works, or the human that is dumping the program from memory, it may be feasible to discreetly insert values in the stack, or library writeable data segments, on the premise that the tool / human wouldn't dump the stack of library data segment, which most (all?) currently don't.

The values inserted may be used for multiple things, such as:

- logic decisions
- modifying data
- decryption keys for other places

As long as there the lack of the data inserted to those locations isn't obvious (for example, it manifests itself a lot later in program execution) it will trip up a lot of people, but once they work out the reliance, the trick doesn't provide much protection.

Restrict available code

See Binary Modifications, specifically Per-page encryption, and Per-function encryption for more information.

Inter-Process Communication

By using Inter-Process Communication (IPC) primitives, such as shared memory, you can setup multiple processes that interact with each other in various ways that can be used to make it harder to understand what is happening.

Implementation of these ideas are left as an exercise to the reader.

General things for consideration

This section of the document is meant to provide a good overview (not necessarily in-depth view) of some subjects that you may wish to consider when writing protection schemes.

Key / data storage

One of the problems that present themselves is how to effectively hide or obscure keys from prying eyes. There are several methods which will be discussed below.

Virtual CPU

The general idea behind a virtual CPU is you write your own CPU (and thus, if you choose, your own assembly language) to handle your own format of assembly commands. This in turn requires the attacker to analyse the virtual CPU operations and work out what is happening inside it.

A pseudo-code of a virtual CPU could look like the following:

```
while(1) {
        switch(*instruction & 0xf) {
                case HALT: break;
                case ADD_REG:
                reg[(*instruction & f0) >> 4] += (instruction[1]) |
                        (instruction[2] << 8) | (instruction[3] << 16)
                        (instruction[4] << 24);
                instruction += 5;
                break;
                ...
        }
}
```

In order to help obscure the operations of the virtual CPU there are various techniques which can be applied to each different product/release, such as:

- The instructions about to be executed configure various parameters of the virtual CPU.

- Swapping opcode mnemonics around. This helps ensure the attacker has to thoroughly analyse the virtual CPU operations so a “disassembler” if needed can be made.
- Don't do “obvious” testing against the parameters, as obvious testing will help a targeted emulator determine what your virtual CPU is doing.

To help prevent analysing one virtual CPU helping to break another similar virtual CPU, here are some general rules you could apply:

- Randomise structure layouts
- Randomise code layouts
- Randomise operators used for testing the values of things.

Taking the virtual CPU idea further, you could simplify the development for it by writing a simple language and compiler for it. Additionally you'd most likely get the benefit of thinking and learning about new things.

Virtual CPU's, if implemented correctly, provide a big challenge for people who wish to attack the binary.

A generalised attack on virtual CPU's:

- An emulator which tracks what parameters you've accessed, and what type of comparisons was made against them.

  For example, if you accessed the register EFLAGS (in the signal handler ucontext variable) and the variable had an AND operation with a 0x100 on it, you could hazard a guess, which would be likely, that its checking whether or not the TRAP flag was set.

  To help defeat this type of emulation attack, you could access multiple variables, change them, and copy them around the place. This makes more work involved for the emulator developers and/or users, perhaps enough so that its not feasible to use this type of attack on the binary.

There has been previous work on using virtual CPU's for protection methods, you can find more information here: [54]

54 http://contests.anticrack.de/index_cpu.htm

Generating the key from the environment

If you know in advance what type of environment the binary is going to be running in, you could generate the keys from it. This has the added benefit that keys aren't stored on the machine or in a file somewhere.

This is somewhat similar to the section about tying binaries to a particular host, and is pretty much only practical when you know what you want to use in advance on that host.

Storing / Getting the keys inside the binary

As opposed to doing *unsigned char key[]* = “*xxx*” in the program, you could use various instruction bytes for the keys and combine them together to make the key. This has the added advantage that you could possibly tell if an attacker has put a software breakpoint on those instructions, or patched the binary.

When obtaining the bytes from memory, it would be a bad idea to use labels to directly access the byte(s), as this will make a data cross reference inside disassemblers, and makes it easier to identify what they are testing.

This additionally allows a limited form of self-checking if used correctly.

Crypto usage

Things to be wary of

If you're going to use cryptography in your programs to protect it (or, just in general), there are several things you should know before you start:

- Stick to publicly analysed algorithms.

Unless you're a cryptographer, don't try and implement your own custom encryption algorithm, as chances are, you'll miss something and your algorithm will have various weaknesses.

By sticking to publicly well-known encryption algorithms, you can be somewhat sure multiple people have analysed them looking for cryptographic flaws. Many cryptographers are suspicious of proprietary encryption algorithm's [55].

- Be wary of known plain-text attacks

  Known plain-text is where the attacker knows some of the plain-text and the cypher-text, and this may allow them to attack the cipher with greater ease.

  An example of known plain-text being of great use is where you have encrypted a function. Most functions will start with several, known bytes (*pushl ebp; movl %esp, %ebp*) and thus can be used to narrow down to what is a likely good key through brute force, or depending on the algorithm used, it may allow the attacker to extract the key used for encrypting the data (think of a small sized constant xor).

Watermarking

Watermarking is the process of inserting information (usually “imperceptibly”, which is defined as per what you are inserting it into) into, or onto, an object, usually inside the signal area for graphics or music. A lot of information about watermarking can be gleamed from [56].

By inserting a watermark, you can track to whom a certain release was distributed to. Of course, this assumes you can have a small enough client base that makes this feasible (i.e., not mass marketed and sold on shop shelves). This may apply if you are producing beta copies for people to use / test, or alternatively if your software has a limited market. If this is the case, you have several opportunities that you can use.

Personalising a copy to them

55 http://www.shmoo.com/mail/cypherpunks/feb99/msg00205.html
56 http://www.watermarkingworld.org

When the software is shipped to them, tell them it is personalised to them. Perhaps, if feasible, send it on a CD/DVD with felt tip pen writing. You can put the company name / user name in the binaries on the CD/DVD, with an appropriate label. If it is a final build of a software product, perhaps a message like “Produced exclusively for *person* (and company, if applicable) on the *date*, build number *#2313123*”

This method is weak, but would help to deter people from just sharing it with their friends.

Proof of ownership

If you needed to identify where a leak came from, you could a small note inside the binary so it can be tracked. There are a couple of methods that can be used to do this.

Simple counter

One such method would be a simple *static watermark_release_date[] = “\x00\x0x\x00\x02”*, whose value would be incremented each time the code is compiled, and then stored away until it was required to identify who leaked the build.

For the most part, these methods suffice if the person looking to leak a copy doesn't know where it is, and just the knowledge that there is a watermark helps deter people from leaking.

Public Key cryptography

Public key cryptography is used in this case to encrypt an identifier for the person to embed in their program. The data is signed using the public key, making it only decryptable by the corresponding private key.

What data you store to track the copy will be dependant on what are the ramifications of it happening, and what you plan on doing about it.

Plain hashing

The plain hashing approach provides a small amount of data (16 to 20 bytes) that can be inserted into a binary that will allow you to identify it. This method is somewhat different to “Simple counter” above and works well in tandem.

To obtain the value you need to insert into the program, do something similar to the following[57]:

```
echo “VERSION released to Person's name” | sha1sum
```

You'll need to remember the string used and the value returned, as this data should (will) uniquely identify everyone who gets a copy.

Out of the two methods shown so far, Public Key cryptography has the advantage that you only need to store your private key to work out who released it.

Storing the watermark

To make you think of how you are going to store the watermarks[58], here is something for you to think about:

If a person wishing to remove the watermark has two different copies of the program and/or data files, how are you going to help prevent the water mark being removed?

For most cases where that isn't really a consideration, then a *fragile* watermark system would be fine, however, when it is a consideration, finding a strong method of watermarking is difficult.

57 Since this particular naming scheme can now be considered public, you are better off picking your own style of labelling it. Adding some known only to you data to the line will help prevent people guessing what you use to label it.
58 To implement this securely, you'd want to have several (possibly different) watermarks.

An example of a fragile watermark would be to embed in a several locations a special string that means something to you. An example of a fragile method would be the *static watermark_release[]* bit mentioned above.

A strong watermark system is going to involve a lot of research, and implementing the watermark everywhere, for example:

- Program logic code
  - A simple example:

    ```
    void do_stuff(int foo, int bar)
    {
           int baz, eeee;

           baz = 58;
           printf(“hihi\n”);
           eeee = foo;
    }
    ```

    versus:

    ```
    void do_stuff(int foo, int bar)
    {
           int baz, eeee;

           printf(“hihi\n”);
           eeee = foo;
           baz = 58;
    }
    ```
- Program image
  - *static unsigned watermark[] = “\xfe\xe1\xde\xad\xca\xfe\xba\xbe”;*
- Program data
  - If you have a complex structure used for controlling how your program works, you could reorder the entries in the structure.
- Images used in the program
  - Preprocessing of the images before release
- Sound used in the program
  - Preprocessing of the sounds before release
- Data / Data files in the program (if possible)
- Output files (if possible.)

Even then, you won't be sure that someone won't be dedicated enough to go through and disrupt / identify / attempt to attack the watermarks scattered over the place, however, it becomes more and more unlikely they'll successfully succeed with that approach, and, perhaps their approach will change, and they'll go after other things, for example, perhaps the source code if feasible.

Here are some notes on implementing the above.

Program code / Function ordering

The files for this section is in watermarking/function_ordering.

The general idea behind function ordering[59] is to link / compile the binary in such a way that the way the binary is laid out tells you who it was released to.

Consider for example you have 4 object files, *main.o, module1.o, module2.o, module3.o*. We can combine the ordering of those files in *!4* (4 x 3 x 2 x 1 == 24) ways to create different binaries.

Using *objdump -d* we can verify that the binary is laid out as we expected. For *binary_1* (compiled by *main.o*, *module1.o*, *module2.o*, and *modile3.o*)

```
...
08048384 <main>:
...
080483a8 <module1>:
...
080483c4 <module2>:
...
080483e0 <module3>:
...
```

Where as on binary_15, we see:

```
...
08048384 <module2>:
...
080483a0 <module1>:
...
080483bc <main>:
...
```

59 Thanks to upb for pointing this method out.

```
080483e0 <module3>:
...
```

To generate these files, all that is needed is to change the arguments passed to the linker. Check the file '*Makefile*' in the source directory. Depending on optimisation levels and the implementation of the linker, it may reorder the functions as it sees fit, so that is something to keep in mind.

As most decent sized programs will have larger amounts of *object* files, this can be used to distribute unique binaries to customers. This could be used in smaller source code files that are distributed by the way the functions are ordered.

There are two methods that can be used to keep track of the binaries. The more data-intensive one is to record what order it was compiled, and to whom it was shipped to.

The other method to is to use the function ordering to encode a value which you can use for tracking purposes, which fits into other systems you already have (such as customer account code / id, their contact details, etc.). To prevent an attacker from working out what method you used, it would be a good idea to encrypt the value with only a key you know. (Said key does not have to be stored in the binary at all, as the encrypted value is calculated before the program is fully compiled.)

Extracting the key can be a bit tricky. One approach to extracting the key would be to work out where in general the functions will lie in memory, and disassemble it manually and recreate the watermark data. Of course, this method is a lot more work.

Another approach to this is to store what bytes you expect to be where, and iterate over the list of fingerprints until you find the best match.

If the binary is sent to the customer stripped, you could use Fenris[60]'s function fingerprinting and symbol dressing features. This would allow you to write a tool to iterate over the symbols list and reconstruct the watermark data.

A more automated method (as in, less steps that need to be performed) of

60 http://lcamtuf.coredump.cx – look for fenris.

extracting the key would be to use IDA and use the function fingerprinting, and write a script to recreate the watermark data. (This is similar to the fenris method, but a different application is used.). What method you use depends on the tools available to you, how much time you have to do it in, and how automated and resistant to attack you want it to be.

Data inside the program

There are various places where seemingly random data will appear inconspicuous in a program. For example, if the program has a pseudo random number generator in it, the random initial values could be part of the key of the program.

This particular method of doing this also means you can functionally prove that the watermark is still there by seeding the random number generator to a certain value, and by using the results generated to preform some operations.

If after the watermark is removed, and a certain amount of time (remember, you don't want to tell the person when they're attacking it straight away that something has changed) the program can then display a file modified message, an obscure message (that they'd ring up support for[61]) or generate incorrect results.

Another area where random data wouldn't look so obvious would be in various initialisation areas, where particular types of data (*u_int8_t*, *u_int16_t*, *u_int32_t*, *char x[6]*, for example), are used. This values can be spread across the program image as needed.

Structure layout

If the program has a complex structure (or several) used for controlling the program, you could order the structure differently for each unique person, and use that to store the watermark.

For a simple example:

61 Remember that modifications may not be caused by deliberate actions on behalf of the user. Other software they use, or a virus, can make those modifications.

```
struct {
        float damage_multiplier;
        int hit_points;
        enum weapons current, backup;
        int score_points;
        struct secret_areas secret_areas[SECRET_NUM];
        int current_level;
        int index_to_structure;
        struct map_layouts current_map;
} game_data;
```

versus:

```
struct {
        struct secret_areas secret_areas[SECRET_NUM];
        int current_level;
        enum weapons current, backup;
        float damage_multiplier;
        int hit_points;
        int index_to_structure;
        int score_points;
        struct map_layouts current_map;
} game_data;
```

The benefit of doing this approach is that it now means that an attacker must identify what types of data is inside the structure, such as pointers, integers, floats, union's and modify the rest of the binary to rearrange the structure, which in a complex application would be a labour intensive undertaking. However, given time and enough need for this, there will be advances, but unlikely any completely automatic solutions.

This method provides a pretty good method of resiliency against someone modifying the binary, due to the amount of work that must go into identifying the structure, and then modifying the binary. Additionally, you could access the data indirectly to increase the amount of time someone has to spend analysing the binary.

For an interesting conversation / exchange of ideas regarding this concept, see footnote [62].

General notes

There are several things that should be kept in mind when considering /

62 http://www.searchlores.org/protec/eceono1.htm

implementing watermarking. Specifically, keep in mind the attacks upon upon watermarking, and how easy it is to apply to compiled binaries, and what counter measures you can apply.

A brief overview of attacks on watermarking are:

- Additive

  Additive attacks attempts to render the watermark unreadable by inserting a new watermark using same/similar methods that the suspected (or known) watermarked binary uses, in the aim of overwriting the existing watermark.

- Distortion

  A distortion attack attempts to remove all places where a watermark could reside, by scrambling the contents (if applicable), usually with a very slight / unnoticeable change to the binary or image..

- Subtractive

  A subtractive attack attempts to erase a watermark.

Some counter measures that can be applied is to have multiple separate and redundant copies of the watermark , and using error correction codes to recover in case of modifications (to a limit.)

It seems that due to the complexity of extensively modifying compiled programs makes it extremely feasible and favourable to insert complex watermarks (such as function ordering, and structure layout, and to an extent, program data) to track who a program was shipped to.

There is a drawback, however, with using function ordering, structure layout, and program data for watermarking, which is that it becomes a lot more difficult to patch or upgrade the program involved (because, obviously, the layout is different each time.). If being able to patch the binary at a later date is not a consideration, then there is no problem using it.

There are some programs that exist already that implement watermarking, one covering source code / English documents [63], and one for binary programs [64].

63 http://lcamtuf.coredump.cx/snowdrop.tgz
64 http://www.crazyboy.com/hydan/

Conclusion

Summary

This document has shown some various methods that can be used in order to make your programs more resistant to analysis by other people, and some methods that can be used to implement strong license number / serial number methods.

Additionally, this document has provided some “food for thought” for those who are implementing such systems, and some exercises the reader can use to fortify their knowledge and understanding.

It is hoped that this document has been useful for the reader, or at least been an entertaining read.

The future / closing thoughts

Due to, what they'd like you do believe, rampant copyright infringement (see below for a mini monologue on the term pirate) of programs, images and multimedia information of companies who don't wish this to happen, companies are pushing to have methods of putting said information on your computer without you being able to copy and analyse the information.

This is understandable, however, there is great potential for this capability to be used against the consumer, as opposed to actually benefiting them. While some people say that it will actually benefit the consumer, without the checks and balances in place, I suspect[65] this is very unlikely.

An example of this being used against the consumer will be when the consumer doesn't have a choice in whether or not the protection method is active on their machine, and whether or not it can be activated without their consent.

65 Its been said I am a very cynical person before.

As opposed to trying to justify to other people / yourself why you use a piece of software / etc without paying for it, don't use it and find another suitable product.

If you don't like commercial software, there are other alternatives, such as Linux, or the BSD's, and all the other types of programs[66] that you can use and modify without paying money, and in some cases, they are superior to the commercial software in what you need and use in that program. If you don't feel like changing operating systems, there are still plenty of programs you can use as a replacement for standard utilities.

In most cases, these programs even allow you to have the source code for their programs, and you're allowed to make modifications and redistribute as per the license[67].

When there is software that doesn't meet your needs, you are welcome to write your own programs to do it, and if you want, release it under these licenses to allow other people to do the same.

Finally, I'd like to point out what the use of the word pirate is very emotive, as pirates are people who rule the seas with terror, rape women, kill men and children, and raid towns and generally cause problems. When emotive terms are used to describe other people, whose actions are nothing like that, they will obviously want to respond by using more emotive action, such as labelling the other people as greedy, and so on.

Due to these emotive terms, and their frequent use, its extremely hard to have a logical discussion about the issues surrounding copyright infringement. It's understandable that people feel this way, however, it doesn't mean that any discussion on the issue has to end with a shouting match.

Feedback and thanks

I would like to say thanks to Raven for the many interesting and informative

66 For example, web browsers, web servers, email clients, office productivity software, mathematical software, etc.

67 Most licenses restrict to a very fine degree what you can do, where as the GPL and BSD style licenses (amongst others) allow you to modify the software and make changes as long as you follow the license information.

discussions we've had on protection systems, assembly and other random things, the people who beta-read this document for me and provided suggestions, the Feline Menace people, Snow for the awesome picture used on the cover page, and you, the reader.

To provide feedback, I can often be found on either SILC[68] or IRC[69], or alternatively, feel free to email me at *andrewg@felinemenace.org*. There are various anti-spam filters set-up on the mail server there. If you don't get a response within a reasonable amount of time (I'll usually respond quickly (being a day or two), but I may be doing other things), check to see if the time on the sending machine is correct, and that your mail server isn't listed on any RBL's.

Additionally, I will most likely be found at RUXCON[70], a computer security conference in Sydney, Australia. You can most likely ask the staff members where I am, or alternatively, email me and arrange some time to meet up.

68 irc.pulltheplug.org, or alternatively, felinemenace.org. Both servers are linked.
69 irc.pulltheplug.org #social or #vortex.
70 http://www.ruxcon.org.au

A brief overview on ELF

What is ELF?

The Executable and Linker Format (ELF) is the binary file layout for most popular UNIX systems and Linux. It is used to represent core files, shared libraries, relocatable objects, and executables.

The reference specifications can be downloaded from [71]. You will most likely need to refer to this later if you'd like more information about what's happening.

On Linux, you can find the C header file for ELF at */usr/include/elf.h*.

A quick breakdown of ELF

Executable Header

The executable header lies at the start of the ELF file. Because the header definitions are the best way of explaining it, it is included in-line.

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

71 http://www.linuxbase.org/spec/refspecs/elf/elf.pdf

To check if a file is an ELF file, you can do memcmp(file_start, ELFMAG, SELFMAG)==0.

To locate either the program header or section header, *lseek()* to the appropriate offset (e_phoff or e_shoff) in the file and read in the appropriate data size.

For example, read(fd, phdr_array, ehdr.e_phnum *ehdr.e_phentsize). You'll want to do various sanity checking if the program is going to be used by other people, such as checking e_phsize is the same as sizeof(Elf32_Phdr) and ensuring various integer overflow possibilities don't happen.

Program Headers

This is where most of the more interesting stuff will happen, as this controls where data is loaded into the memory space. The program headers are defined as:

```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```

Loading program headers can be non-obvious at first, especially when you see non page-aligned load addresses.

For example, on the authors system, /bin/ls has these load headers:

```
LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
     filesz 0x00011d08 memsz 0x00011d08 flags r-x
LOAD off    0x00012000 vaddr 0x0805a000 paddr 0x0805a000 align 2**12
```

This shows that the second (.data section) is page-aligned on the disk and in memory. However, /bin/ps has these headers.

```
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000f788 memsz 0x0000f788 flags r-x
    LOAD off    0x0000f788 vaddr 0x08058788 paddr 0x08058788 align 2**12
```

To work out how to load this section is relatively simple, and is covered in "Loading executes in user-space".

When the page size and memory size of the header doesn't line up, it means there are some variables stored in the *.bss* section, which isn't stored in the file and is initialised to 0.

Section Headers

The Section Headers "describe" an ELF file, and aren't needed to load the file into memory.

Some of the things the section headers will define are:

- What functions are defined in the program, their name and length.
- The string table of functions, section header names, etc.
- Where various pieces of data are, such as constructors, destructors, where the data section starts, where the bss starts etc.

The section header is defined as:

```
typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

The usually interesting sections of this for us are:

```
#define SHT_SYMTAB        2             /* Symbol table */
#define SHT_STRTAB        3             /* String table */
```

Usually speaking, the last two entries of the section headers will be the ones we are after. These values hold respecting the symbol table (which defines such things as where functions start and how long they are, and whether or not something is a data section, and how long it is, etc) and the name table, which gives you the names of what those previous things where defined as.

The symbol table is defined by:

```
typedef struct
{
  Elf32_Word    st_name;                /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;               /* Symbol value */
  Elf32_Word    st_size;                /* Symbol size */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf32_Section st_shndx;               /* Section index */
} Elf32_Sym;
```

The string table can be cast to a char *, and to find out the name of a symbol, use *string_table + st_name* to get the name.

An example of parsing the section headers can be found in the Per function encryption section in Binary modification.

Mammon's gdbinit file display

This section is meant to provide a brief run down of mammon's gdbinit file, and the data you're looking at when your program stops executing.

Doing *gdb /bin/ls*, and typing *sstart* gives the following (colours added added for emphasis):

```
gdb> sstart
Breakpoint 1 at 0x400486f0
<snip>
_______________________________________________________________________________
     eax:00000000 ebx:40016C00  ecx:BFFFFAD4  edx:4000C290     eflags:00000246
     esi:00000001 edi:08049A50  esp:BFFFFAAC  ebp:00000000     eip:400486F0
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t s Z a P c
[007B:BFFFFAAC]---------------------------------------------------------[stack]
BFFFFADC : DB FB FF BF  EE FB FF BF - FE FB FF BF  09 FC FF BF ................
BFFFFACC : 00 00 00 00  01 00 00 00 - D3 FB FF BF  00 00 00 00 ................
BFFFFABC : A0 61 05 08  00 62 05 08 - 90 C2 00 40  CC FA FF BF .a...b.....@....
BFFFFAAC : 71 9A 04 08  A0 9E 04 08 - 01 00 00 00  D4 FA FF BF q...............
[007B:08049A50]---------------------------------------------------------[ data]
08049A50 : 31 ED 5E 89  E1 83 E4 F0 - 50 54 52 68  00 62 05 08 1.^.....PTRh.b..
08049A60 : 68 A0 61 05  08 51 56 68 - A0 9E 04 08  E8 F3 FC FF h.a..QVh........
[0073:400486F0]---------------------------------------------------------[ code]
0x400486f0 <__libc_start_main>: push   %ebp
0x400486f1 <__libc_start_main+1>:       push   %edi
0x400486f2 <__libc_start_main+2>:       push   %esi
0x400486f3 <__libc_start_main+3>:       push   %ebx
0x400486f4 <__libc_start_main+4>:       sub    $0x4c,%esp
0x400486f7 <__libc_start_main+7>:       mov    0x64(%esp),%eax
------------------------------------------------------------------------------
0x400486f0 in __libc_start_main () from /lib/tls/libc.so.6
gdb>
```

The blue section is the display of the current registers, and a part of *eflags* which is used to make program logic decision, or that a particular event has happened. The bottom set of registers (cs, ds, es, fs, gs, and ss) and effectively be ignored for most cases under Linux.

When a letter out of the eflags display is capitalised, it means the bit is set in eflags. Conversely, when it is lower case, it means the bit is not set.

In this example, we can see that the *Interrupt flag*, *Zero flag*, and *Parity flag* is set. To find out more about these flags means, consult the Intel documentation, or the nasm documentation.

The green section refers to the programs current stack layout, along with any printable ascii characters. Remember that the IA32 platform is little-endian, which means to get the first word off the stack, we need to reorder its meaning. If we read "*DB FB FF BF*" backwards, we see that it is 0xBFFFFBDB.

The yellow section refers to an applications "data" area, which is determined by checking the registers *edi*, *esi*, *eax*, and finally falling back to esp if it can't find a register that has its MSB pointing to a memory mapped area, programs data section, or stack.

The red area refers to the programs current instructions it will be when the program execution is continued.

Hopefully this clears up some things to people new to this gdb configuration file.