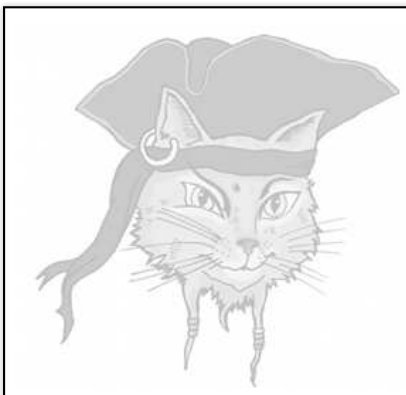

Surreptitious Software

Exercise

Attacks
Watching Data

Christian Collberg
Department of Computer Science, University of Arizona
February 26, 2014

Watching data

p. 76

A common software protection technique is to make the program crash when it detects that someone is trying to tamper with it. For example, the program can set a pointer to `NULL` causing a segmentation fault when the pointer is dereferenced.

```
int* ptr = ...;
     ⋮
if (I-am-tampered-with) {
   ptr = NULL;        ⇐ tamper-response here!
}
     ⋮
x = *ptr              ⇐ crash here!
```

For an attacker, a crash can provide useful information. He can run the program until it crashes and then trace back from the crash site to the location that caused the crash.

The Program

Figure 1 shows the code for `player3`. Notice how the detection of the wrong code and the setting of `player_key` to `NULL` happens in one place, and the dereference of `player_key` in another! This makes it harder for the attacker to trace back from the segmentation fault to the cause of the crash.

The Algorithm

In our `player` program, the tamper-response is to set the `player_key` variable to NULL. The next time it is referenced, a segmentation fault is thrown.

So, our strategy here is as follows:

1. Run the program until it crashes,
2. see what pointer variable was dereferenced,
3. set a watchpoint on that variable,
4. run the program again,
5. when the watch-point is hit, we know the location where the variable was set to NULL,

```
uint32 the_player_key = 0xbabeca75;
uint32* player_key = &the_player_key;
FILE* audio;

int activation_code = 42;
uint32 play(uint32 user_key, uint32 encrypted_media[], int media_len) {
   int code;
   printf("Please enter activation code: ");
   scanf("%i",&code);
   if (code!=activation_code) {
      fprintf(stderr,"%s!\n","wrong code");
      player_key = NULL;                    ⇐ Tamper-response here!
   }
   int i;
   for(i=0;i<media_len;i++) {
      uint32 key = user_key ^ *player_key;  ⇐ Crashing here!
      uint32 decrypted = key ^ encrypted_media[i];
      float decoded = (float)decrypted;
      fprintf(audio,"%f\n",decoded); fflush(audio);
   }
}
```

Figure 1: The code.

6. replace `var = NULL` with `NOP`s.

Prerequisites

Before working the exercise make sure you download, install, and build the following:

1. Install the following tools:

tool	url	Linux	MacOS X	Windows
gcc		✠ `gcc` `build-essential`		
gdb	`ftp.gnu.org/gnu/gdb/`	✠ `gdb`		

2. Download program and data files:
   (a) `wget 'http://www.cs.arizona.edu/~collberg/tmp/ssx.zip'`
   (b) `unzip ssx.zip`
   (c) `cd ssx/attack-defense_attack3`
3. Build the `player3` executable which you will be working on from now on:

```
> make
```

Crack — Remove the tamper-response!

1. Run the program, enter the wrong activation code, and wait for the program to crash.

```
(gdb) run 0xca7ca115 1000 2000 3000 4000
Please enter activation code: 43
wrong code!
```

What is the address where the pointer variable is dereferenced?

2. Disassemble a region around this point to see where the address was computed. Show the instructions here:

3. Looking at these instructions, can you figure out what the address of the variable (`player_key`) is? Write it here:

4. Now, set a hardware write watchpoint on the address of the pointer variable. Show the `gdb` command:

5. Re-run the program with the watch-point set. With some luck, `gdb` will now tell you where in the program the variable gets set to null! Show the address!

NOTE: If `gdb` doesn't say "`Hardware watchpoint 1: ...`" it means this particular installation doesn't support hardware watchpoints. The result is that execution will be very slow. `gdb 7.2` on Mac OS X, for example, doesn't seem to support hardware watchpoints — use `gdb 6.8` instead.

6. Disassemble a bit of code around this address until you see the `movl` instruction that stores 0 into the pointer! Show the instructions here:

7. Now, based on the instructions you just saw, what's the address of the instruction that sets `player_key` to null? How long is the instruction?

8. Now quit `gdb`, and start it up again.

   Next, write over the `movl` instruction with something innocuous! The `NOP` instruction is 1 byte long and has the opcode `0x90`. How many of them do you need to insert to obliterate the `movl` instruction? Show the `gdb` instructions here:

9. Disassemble the region again!

10. Exit `gdb`! Is the program working?!?! Use `vbindiff` to compare the old and the new versions!