

CSc 466/566 Computer Security

Assignment 3

Due Noon, April 11, 2012
Worth 10% (ugrads), 5% (grads)

Christian Collberg
Department of Computer Science, University of Arizona

Copyright © 2012 Christian Collberg

1. Introduction

In this assignment we'll study buffer overflow attacks and man-at-the-end attacks.

- This is an individual assignment. You cannot collaborate with anyone or get help from any human.

2. Buffer Overflow Attacks

2.1 Setting up the environment

1. Go to

http://www.cis.syr.edu/~wedu/seed/lab_env.html

and download

`SEEDUbuntu9_August_2010.tar.gz`.

This file contains an image of the virtual machine on which you should work. The homework will be graded only on this virtual machine.

2. Download VMWare player from

<http://www.vmware.com/products/player>

and install it on your workstation.

3. Open VMWare player and navigate to

`File→Open a Virtual Machine`

and open the VM downloaded in step 1.

4. To login to the system, use the following credentials:

Username : `seed`

Password: `dees`

Certain steps in the homework will require you to have root access. In such cases, use the following credentials:

Username: root
Password: seedubuntu

In case of any difficulties, start by consulting the user manual for the VM:

http://www.cis.syr.edu/~wedu/seed/Documentation/Ubuntu9_VM/Ubuntu9_VM_Manual.pdf

2.2 Tasks to be performed

Go to

/50

http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Buffer_Overflow

and download the instructions for the exercise along with the relevant C files. Follow the instructions, and hand in two versions of the files `stack.c` and `exploit.c`:

1. Create two folders `task1` and `task2`.
2. In each folder, create `task(n)/stack.c` and `task(n)/exploit.c`, corresponding to the two tasks.
3. Create a file `report.pdf` describing how you solved tasks 1-4.
4. Zip up `task1`, `task2`, and `report.pdf` in a zip-file called `buffer.zip` and submit this to d2l.

NOTE: Task 2 has some bonus points — these questions are mandatory!

3. Man-At-The-End Attacks

`player` is a digital rights management program. You call it like this:

```
> player userkey sample1 sample2 sample3
```

where `userkey` is a 32-bit cryptographic key and the samples are integers that you want to “play”. In actuality, all that happens is that decode samples are written to the file `audio`. Example:

```
> player 0xca7ca115 10000 20000 30000 60000
Please enter activation code: 42
> cat audio
3133074688.000000
3133047808.000000
3133062912.000000
3133022208.000000
```

Figure 1 shows a block diagram of the DRM player. Figure 2 shows (part of) the actual C code.

- You can download the executable from here:

<http://www.cs.arizona.edu/~collberg/Teaching/466-566/2012/Assignments/index.html>

The executable is compiled to run on `lectura`.

- Read Appendix A to learn how to use `gdb` and other related tools.
- Hand in this (hardcopy) document with the boxes filled in with your answers, or submit the corresponding text file.

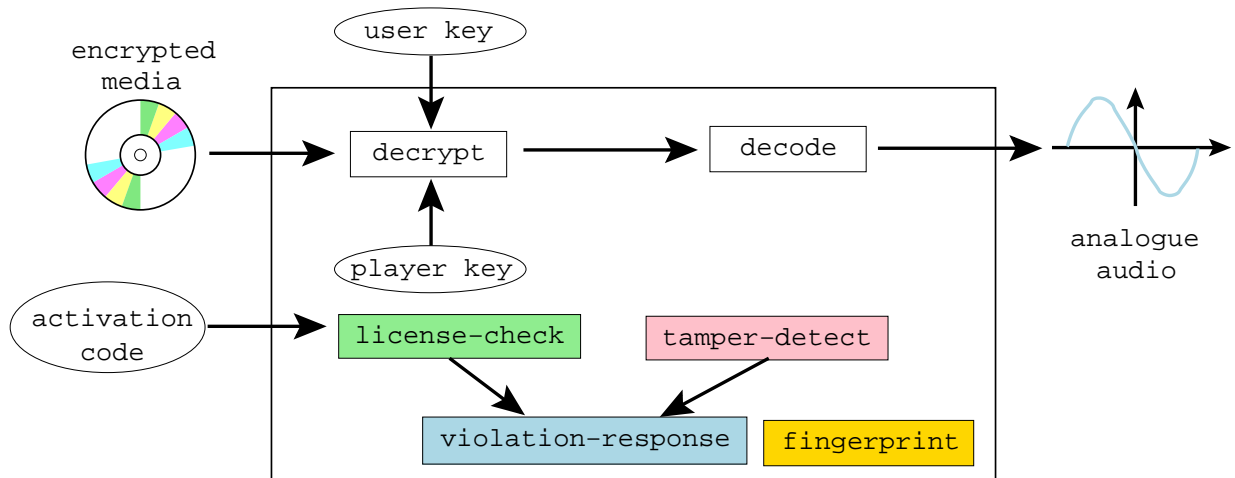


Figure 1: Block diagram of the player.

3.1 Find information about the program!

1. Use the `file` command to find out what kind of executable we're dealing with.

/10

2. Use `objdump` to find the program's entrypoint.

3. Use `objdump` to find the beginning of the text segment.

```

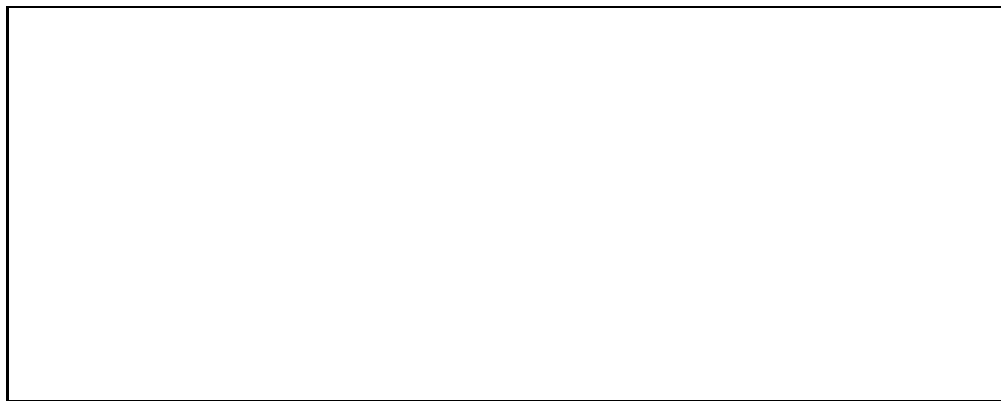
typedef unsigned int uint32;
typedef char* caddr_t;
typedef uint32* waddr_t;
uint32 the_player_key = ...;
FILE* audio;
int activation_code = 42;
uint32 play(uint32 user_key, uint32 encrypted_media[], int media_len) {
    int code;
    printf("Please enter activation code: ");
    scanf("%i",&code);
    if (code!=activation_code) {
        fprintf(stderr,"%s\n","wrong code");
        ...
    }
    int i;
    for(i=0;i<media_len;i++) {
        uint32 key = user_key ^ the_player_key;
        uint32 decrypted = key ^ encrypted_media[i];
        if (time(0) > ....) { ... }
        float decoded = (float)decrypted;
        fprintf(audio,"%f\n",decoded); fflush(audio);
    }
}
uint32 player_main (uint32 argc, char *argv[]) {
    uint32 user_key = atoi(argv[1]);
    int i;
    uint32 encrypted_media[100];

    for(i=2; i<argc; i++)
        encrypted_media[i-2] = atoi(argv[i]);
    int media_len = argc-2;

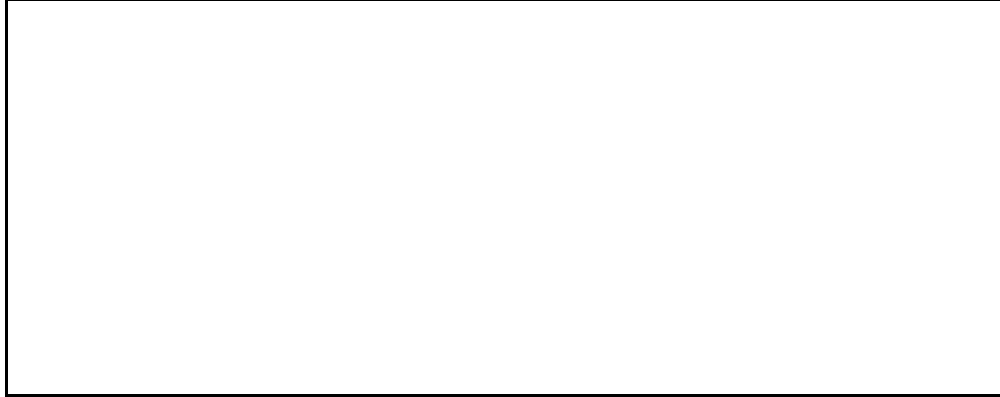
    play(user_key,encrypted_media ,media_len);
}

```

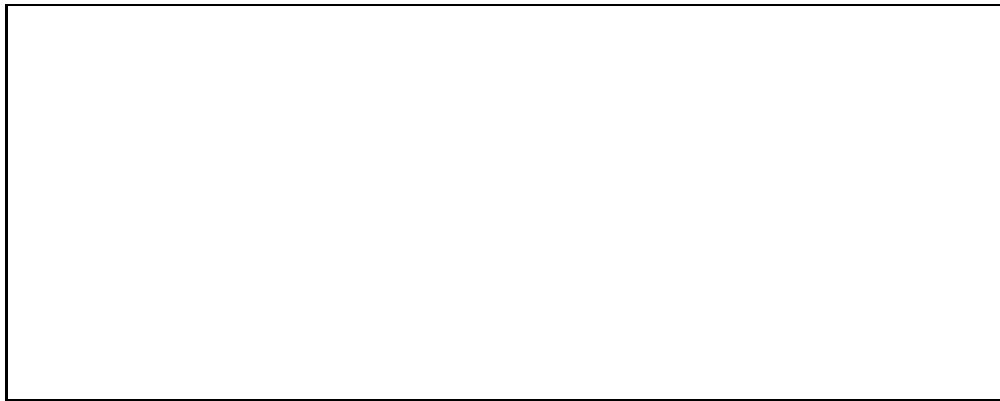
Figure 2: The code.



4. Use `objdump` to find the beginning of the *read-only* data segments.



5. Use `objdump` to find out which symbols the executable has defined.



3.2 Breaking on system function

The `player` fails when its use-by date has been exceeded:

/20

```
> player 0xca7ca115 10000 20000 30000 60000
Program expired!
Bus error
```

This is what the protection code looks like:

```
if (time(0) > ...) {
    ...
}
```

3.2.1 Algorithm — Breaking on system function

We already know that the executable is *dynamically linked*. This means that many library functions can be easily found by name. Most likely, the program calls the `time()` function in the standard library and compares the result to a predefined value. So, the idea we're going to use is to

1. set a breakpoint on `time`,
2. run the program until the breakpoint is hit,
3. go up one level in the call stack (to see who called `time`),
4. look at the assembly code in the vicinity of the call to `time` for the equivalent of

```
if (time(0) > some value)...
```

Table 1: X86 condition codes. Taken from <http://courses.ece.uiuc.edu/ece390/resources/opcodes.html>.

CCCC	Name	Means
0000	O	overflow
0001	NO	Not overflow
0010	C/B/NAE	Carry, below, not above nor equal
0011	NC/AE/NB	Not carry, above or equal, not below
0100	E/Z	Equal, zero
0101	NE/NZ	Not equal, not zero
0110	BE/NA	Below or equal, not above
0111	A/NBE	Above, not below nor equal
1000	S	Sign (negative)
1001	NS	Not sign
1010	P/PE	Parity, parity even
1011	NP/PO	Not parity, parity odd
1100	L/NGE	Less, not greater nor equal
1101	GE/NL	Greater or equal, not less
1110	LE/NG	Less or equal, not greater
1111	G/NLE	Greater, not less nor equal

and replace it with

```
if (time(0) <= some value)...
```

3.2.2 Crack — Remove the use-by check!

So, let's go ahead and remove the pesky check that makes the program say `Program expired!` instead of playing music for us!

1. Start the `player` program under `gdb`:

```
> gdb -write -silent player
```

2. Set a breakpoint on the system `time` function.

```
(gdb) break time
```

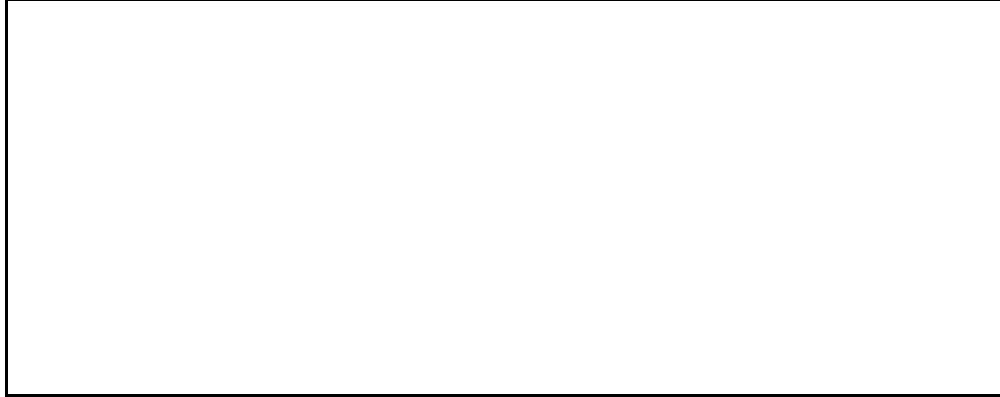
3. Start the program by typing the command

```
(gdb) run 0xca7ca115 10000 20000 30000 60000
```

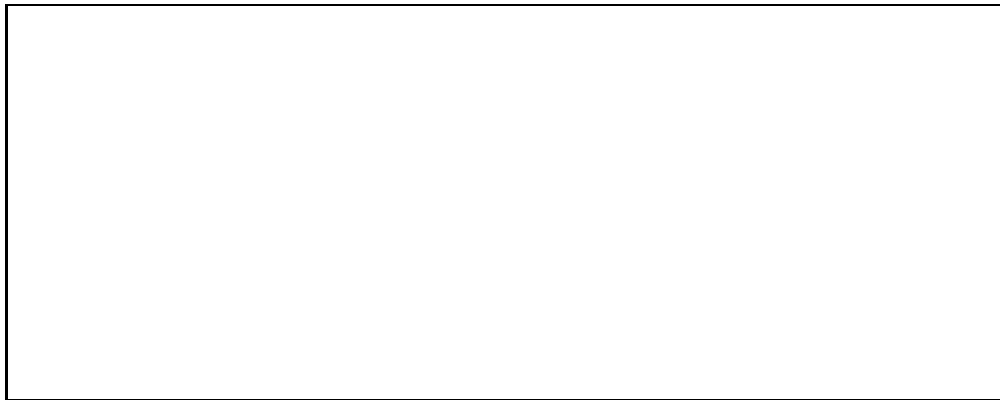
`0xca7ca115` is the secret key. `10000 20000 30000 60000` are the input “samples” to the program.

4. What location is the `time` library function called from? Use the `where` command!

Next use the `up` command to walk up the caller's stack frame and `x/i $pc` to find the address of the current instruction.



5. Find the location where the value `time` returns is tested and the branch that follows the test!



6. The `jle` instruction is two bytes long (how can you tell?). What's the value of these two bytes (in hex)?



7. The second four bits of the `jle` opcode is the condition code. See Table 1 for a list of the X86 processor's condition codes. You now need to invert the branch from a less-than-or-equal to a greater-than! What should the X86 instruction be, in hex?



8. Now you know the location to patch at and what the new instruction should be! It's time to do the actual patch! Start by quitting `gdb`, and then re-entering `gdb`.

NOTE: gdb is really picky about this — you *have to* start gdb from a “clean slate” before you edit the executable or the changes won’t actually affect the executable file.

Show the gdb instructions you used:

```
(gdb) quit  
> gdb -write -silent player
```

```
do the patch here!  
(gdb) quit
```

9. Exit gdb. Run `player`. Does it behave better now?

10. Compare the new `player` with the original one using `vbindiff` which can be downloaded from here:

<http://www.cjmweb.net/vbindiff/>

```
> vbindiff player player.orig
```

Can you find the difference?

3.3 Searching the binary

The `player` program fails if you have the wrong activation code:

/20

```
> player 0xca7ca115 10000 20000 30000 60000  
Please enter activation code: 99  
wrong code!  
Bus error
```

(The real activation code is "42".) Obviously, you want the program to play without nagging you about the activation code!

3.3.1 Algorithm — Cracking by Searching the Binary

To remove the activation code check we’re going to use a slightly different strategy. The latest version (7.0 and later) of `gdb` has the ability to search for a string within the executable.

We can assume that the protection code looks something like this:

```
addr1: "wrong code"  
.....  
read_value = scanf()  
if (read_value != activation_code)  
    addr2: call printf(addr1)
```


or, in pseudo assembly code:

```
addr1: .ascii "wrong code"
.....
      mov read_value, reg0
      mov activation_code, reg1
      cmp reg0,reg1
      je somewhere
addr2: mov addr1, reg0
      call printf
```

So, we

1. search for *addr1*, the address of the string "wrong code",
2. search for *addr2*, the address where `printf` is called,
3. look backwards in the code until we find the instructions that do the check if (`activation_code != 42`) `...`, and
4. patch the code as in the previous exercise.

NOTE: This will only work if the compiler generates *addr1* directly in the code! Some compilers will instead load *addr1* as an offset from a base register — then we can't find *addr2* as easily as this! On Mac OS X we can compile with `gcc -mdynamic-no-pic` to turn off this behavior.

3.3.2 Crack — Remove the activation code check!

Now carry out the attack:

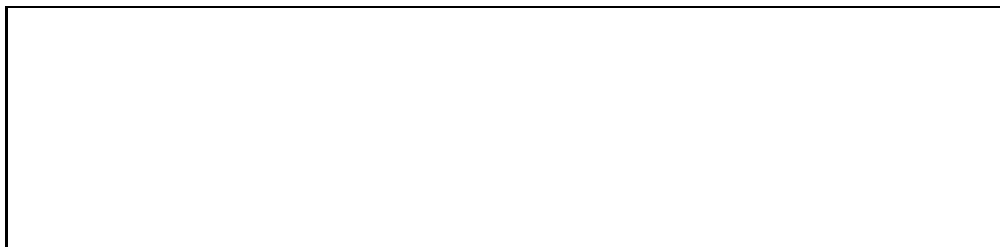
1. Before we can start searching the binary, you need to find out where the text segment and the read only data segment start, and how long they are:

```
> objdump -x player | egrep 'text|Name'           # Linux
> objdump -x player | egrep 'rodata|Name'

> otool -l player | gawk '/__text/,/size/{print}' # Mac OS X
> otool -l player | gawk '/__cstring/,/size/{print}'
```



2. Use `gdb`'s `find` command to find the address of the string "wrong code".¹



¹Be careful to enter the search string exactly; `gdb`'s `find` command doesn't search for partial strings.

Let's call this *data* address *addr1*. Check that you have the right address:

```
(gdb) x/s addr1 here!
addr1:      "wrong code"
```

3. Now use the `find` command again, looking through the *text segment* for an instruction that uses the *addr1* address!

Let's call this *code* address *addr2*.

4. Disassemble a region prior to *addr2* to verify that is the region you're looking for.

5. Now that you know both *addr1* and *addr2*, it's time to do the patching! First exit, and then re-enter `gdb`.

You now need to replace the `je` (jump equal) branch with a `jmp` (jump always). The opcode for `jmp` is `0xeb`. Show the `gdb` command here:

```
(gdb) quit
> gdb -write -silent player1

do the patch here!
(gdb) quit
```

(See <http://www.itis.mn.it/linux/quarta/x86/jmp.htm>.)

6. Try the patched program! Does it work for any activation code?

A. MATE Techniques

A.1 Learning about the executable (Linux)

1. `objdump` prints out information about an executable file. It has lots of options, depending on what you want. The `-T` option prints the dynamic symbols:

```
> objdump -T player2
DYNAMIC SYMBOL TABLE:
00000000      DF *UND* 00000039  GLIBC_2.0  printf
00000000      DF *UND* 0000002b  GLIBC_2.0  atoi
00000000      DF *UND* 00000024  GLIBC_2.0  fprintf
00000000      DF *UND* 00000020  GLIBC_2.0  time
```

2. `objdump` can also disassemble:

```
> objdump -d player2 | head

080483f4 <.init>:
80483f4:      55                push   %ebp
80483f5:      89 e5             mov    %esp,%ebp
80483f7:      83 ec 08         sub   $0x8,%esp
```

3. `objdump` gives you the start address:

```
> objdump -f player2 | grep start
start address 0x080484f0
```

4. `objdump` gives you the address and size of the string (read-only data) and text segments:

```
> objdump -x player2 | egrep 'rodata|text|Name'
Idx Name      Size      VMA      LMA      File off  Algn
 11 .text      00000508  080484f0  080484f0  000004f0  2**4
 13 .rodata    00000075  08048a14  08048a14  00000a14  2**2
```

A.2 Learning about the executable (Mac OS X)

On Mac OS X we have to use `otool` instead of `objdump` for some operations.

1. To print the dynamic symbols:

```
> objdump -T player2
```

2. To disassemble:

```
> otool -t -v player2
```

3. To get the start address:

```
> otool -t -v player2 | head
```

4. To get the address and size of the string and text segments:

```
otool -l player2 | gawk '/__text/,/size/{print}'
otool -l player2 | gawk '/__cstring/,/size/{print}'
```

NOTE: `otool` sometimes displays addresses like “00000bd0”, and sometimes like “0000000100000bd0.” Inside `gdb` use “0x100000bd0” since this is the actual virtual address. (Avoid leading zeros since this indicates an octal address.)

A.3 Tracing the executable

1. `ltrace` traces library calls:

```
> ltrace -i -e printf player2
[0x804884e] printf("hash=0x%x\n", 0x478a1c90hash=0x478a1c90) = 16
tampered!
[0x8048702] printf("Please enter activation code: ") = 30
Please enter activation code:
```

2. `strace` traces system calls:

```
> strace -i -e write player2
[110425] write(1, "hash=0x478a1c90\n", 16hash=0x478a1c90) = 16
[110425] write(2, "tampered!\n", 10tampered!) = 10
[110425] write(1, "Please enter activation code: ",... ) = 30
```

A.4 Gdb

1. To start `gdb`:

```
gdb -write -silent --args player2 0xca7ca115 1000 2000 3000 4000
```

2. The latest version of `gdb` (7.0 and above) has the new `find` command which searches for a string in an executable:

```
(gdb) find startaddress, +length, "string"
(gdb) find startaddress, stopaddress, "string"
```

NOTE: Note that you have to give the *entire* string you're looking for — `find` doesn't do partial searches. I believe it looks for the string *including the null character at the end*, so any trailing spaces, tabs, etc. have to be included in the search.

You can also search for bytes, words, etc.

3. To set a breakpoint at a particular address:

```
(gdb) break *0x.....
(gdb) hbreak *0x.....
```

`hbreak` sets a hardware breakpoint which doesn't modify the executable itself.

NOTE: Note that on `x86-64`, the program must be started before you can set a hardware breakpoint!

4. To set a watchpoint at a particular address:

```
(gdb) rwatch *0x.....
(gdb) awatch *0x.....
```

`rwatch` only checks for reads of the location.

NOTE: Note that on `x86-64`, the program must be started before you can set a hardware watchpoint!

5. To disassemble instructions:

```
(gdb) disass startaddress endaddress
```

or, if you only want to see a certain number (here, 3) of instructions:

```
(gdb) x/3i address
(gdb) x/i $pc
```

The second command prints the instruction at the current address,

6. To examine a data word (**x=hex**,**s=string**, **d=decimal**, **b=byte**,...):

```
(gdb) x/x address
(gdb) x/s address
(gdb) x/d address
(gdb) x/b address
```

You can hit return multiple times to examine consecutive locations.

7. To print register values:

```
(gdb) info registers
```

8. To examine the callstack:

```
(gdb) where
(gdb) bt          -- same as where
(gdb) up          -- previous frame
(gdb) down       -- next frame
```

9. To step one instruction at a time:

```
(gdb) display/i $pc
(gdb) stepi
(gdb) si          -- same as stepi
(gdb) nexti      -- like step, but don't step into functions
(gdb) ni         -- same as nexti
```

The `display` command only has to be set once. It makes sure that `gdb` prints the instruction it's stepping over.

10. To modify a value in memory:

```
(gdb) set {unsigned char}address = value
(gdb) set {int}address = value
```

A.5 Patching executables with `gdb`

Cracking an executable proceeds in these steps:

1. find the right address in the executable,
2. find what the new instruction should be,
3. modify the instruction in memory,
4. save the changes to the executable file.

This process is called *patching*.

`gdb` can patch the executable for us, but it is very picky about how to go about it. There are two ways to start the program to allow patching:

method 1:

```
> gdb -write -q player1
```

method 2:

```
> gdb -q player1
(gdb) set write
(gdb) exec-file player1      # reload the file!
```

`gdb` doesn't allow us to patch the executable when it is running. It's therefore best to:

1. Run the program under `gdb` and find the address of the instruction you want to patch.
2. Exit `gdb`.
3. Start `gdb` again using one of the two methods above.
4. Make the patch and exit:

```
(gdb) set {unsigned char} 0x804856f = 0x7f
(gdb) quit
```