Surreptitious Software

Exercise

Attacks
Breaking on System Functions

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

Introduction

`player1` is a digital rights management program. You call it like this:

```
> player1 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:

```
> player1 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 the actual C code.

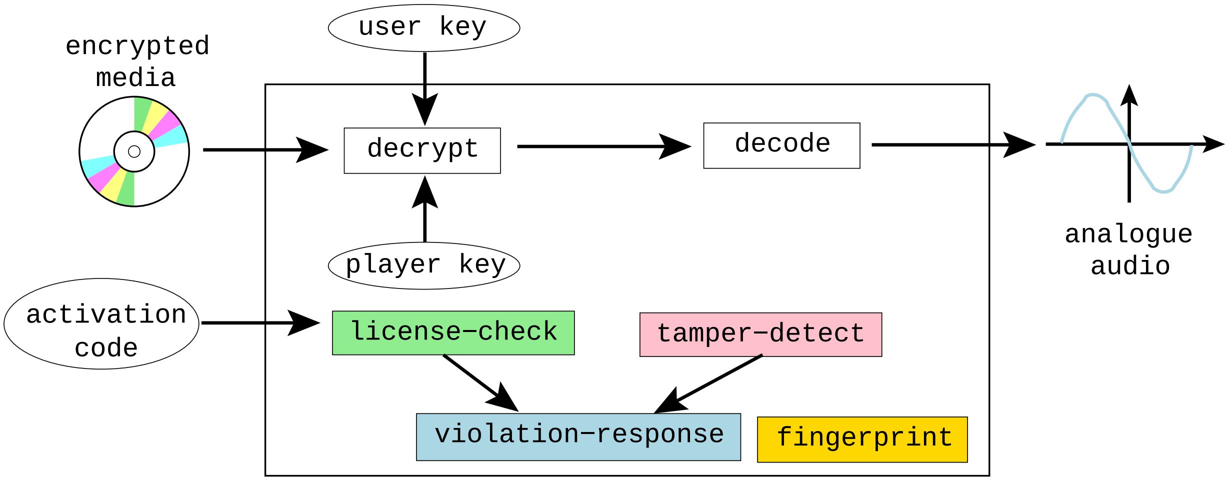

Figure 1: Block diagram of the player.

```
typedef unsigned int uint32;
typedef char* caddr_t;
typedef uint32* waddr_t;

uint32 the_player_key = 0xbabeca75;
FILE* audio;

uint32 play(uint32 user_key, uint32 encrypted_media[], int media_len) {
   int 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) > 1221011472) {
         fprintf(stderr,"%s!\n", "Program expired!");
         *((int*)NULL)=99;
      }
      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);
}

int main (uint32 argc, char *argv[]) {
   printf("This is player1. Usage: player1 0xca7ca115 10000 20000 30000 60000\n");
   audio = fopen("audio", "w");
   player_main(argc,argv);
   return 0;
}
```

Figure 2: The code.

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_attack1

3. Build the `player1` executable which you will be working on from now on:

   ```
   > make
   ```

Software protections

The `player1` program has one simple software protection built in. It also fails when its use-by date has been exceeded:

```
> player1 0xca7ca115 10000 20000 30000 60000
Program expired!
Bus error
```

In future exercises we will add more interesting protection techniques!

This is what the protection code looks like:

```
if (time(0) > 1221011472) {
   fprintf(stderr,"%s!\n", "Program expired!");
   *((int*)NULL)=99;
}
```

Algorithm — Breaking on system function

p. 73

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) > @{\em some value})\ldots@
   ```

   and replace it with

   ```
   if (time(0) <= @{\em some value})\ldots@
   ```

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. Build and start the `player1` program under `gdb`:

   ```
   > make
   > gdb -write -silent player1
   ```

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

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

```
(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!

   Nest 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. Now The 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 player1



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

9. Exit `gdb`. Run `player1`. Does it behave better now?

10. Compare the new `player1` with the original one:

    ```
    > vbindiff player1 player1.orig
    ```

    Can you find the difference?