Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Operating Systems (TI2726-C)
Lab Session 5: Security

D.E. Schut, M. van de Ruit, O. W. Visser & J.S. Rellermeyer

March 22, 2018

Deadline: March 29 2018

Introduction

Session 5 of this lab allows you to explore system security issues. You are asked to exploit a simple application to execute a sensitive function and in the second part you will write a secure loader that only allows signed programs to be executed.

5.1 Designing a Stack Exploit

Familiarize yourself with stack exploits, e.g., by reading *Smashing The Stack For Fun And Profit* `http://insecure.org/stf/smashstack.html`. You will learn about different techniques that can be used to design an exploit for a vulnerable program.
Write a stack exploit for the following program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define BUF_SIZE 256

int main(int argc, char **argv)
{
  char buf[256];
  if(argc < 2)
  {
    fprintf(stderr, "Usage: stack <some input>\n");
    exit(1);
  }

  strcpy(buf, argv[1]);

  return 0;
}
```

```
void hidden_function(void)
{
   fprintf(stderr, "This program has been exploited\n");
}
```

Analyze why exactly the above program is vulnerable, then create and implement a strategy for an exploit. A simple solution requires disabling stack protection when compiling the program with GCC by setting the following option:

```
gcc -z execstack
```

More advanced (return-oriented programming) approaches might not need it. The expected outcome of your exploit is that without any modifications to the program and just by applying your exploit the hidden function is invoked.
For testing, you should compile to program without optimization so that the hidden function does not get eliminated by the compiler.
A reference of ARM assembly can be found here: `http://infocenter.arm.com/help/topic/com.arm.doc.qrc00011/QRC0001_UAL.pdf`.

5.2 Writing a Secure Loader

In this exercise you are asked to write a secure program loader that only executes programs which have been signed and rejects programs with missing or wrong signature.
In order to do so, you first have to create an RSA public/private key pair. In the first step, you create the private key:

```
openssl genpkey -algorithm RSA -out private_key.pem
  -pkeyopt rsa_keygen_bits:2048
```

This key needs to remain a secret. Set the file system access permissions so that only you can read it:

```
chmod 400 private_key.pem
```

Now you can extract the public key from the private key:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

The idea behind secured programs is to create a secure digest (hash) using `openssl` and embedding the hash into the binary as an ELF section named `.sha`. Your loader can now, with the knowledge of the public key which can be made available for verification purposes by the author of the binary, verify the integrity of the program binary before executing it. Binaries that do not contain a valid secure digest or no digest at all should be rejected by the loader.
Figure 1 shows the intended sequence of operations. Please note that adding a header to the ELF binary is a modification and therefore the header needs to be removed again before verification. The intended usage pattern of the loader is the following:

```
loader <secured binary> <public key>
```

Useful tools for this exercise are `openssl dgstl` and `objcopy`. your RaspberryPi image contains the libssl and libelf header files so that you can use the APIs directly.
You can download an example secured binary and matching public key file from Brightspace for testing purposes. The binaries contain a SHA-256 signature, you need to consider this when writing your loader.

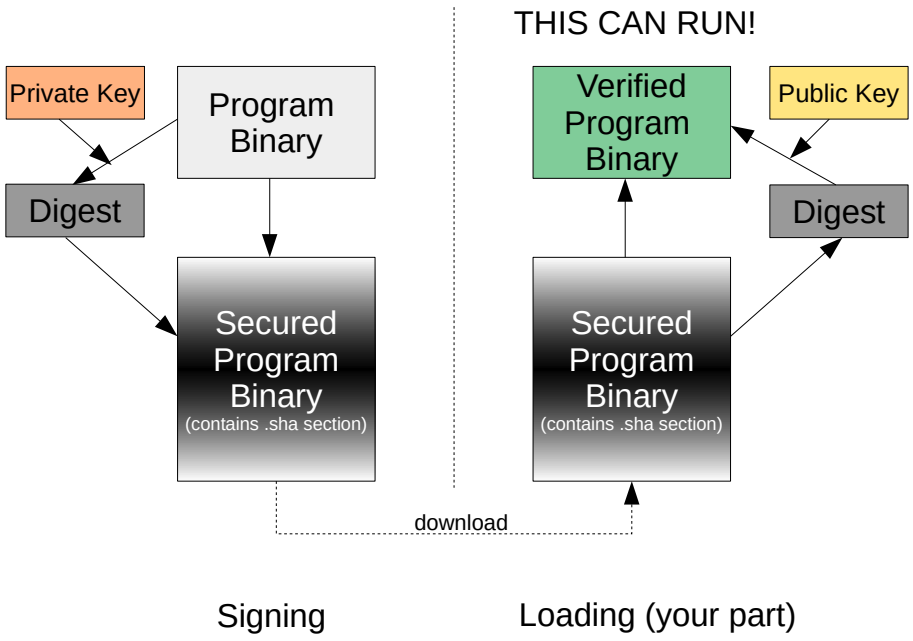

Figure 1: Signing and loading secured binaries.

Bonus

Design an exploit for the program in 5.1 that invokes a shell and thereby hijacks the privileges of the process.