Security 101

Ben Mezger

Univali
`http://www.univali.br`

<2015-08-08 Sat>

Outline

Section 1

Introduction

Cypherpunks

Privacy is necessary for an open society in the electronic age.
We cannot expect governments, corporations, or other large, faceless organizations to grant us privacy out of their beneficence.
We must defend our own privacy if we expect to have any.
Cypherpunks write code. We know that someone has to write software to defend privacy, and since we can't get privacy unless we all do, we're going to write it.[1]

[1] `http://www.activism.net/cypherpunk/manifesto.html`

Julian Assange

- Wikileaks
  - Chief-in-chief
  - Stablished in 2006
  - Published classified secret information
- Freedom of speech
  - Believes governamental documents should be public
- Programming
  - Author of Transmission Control Protocol (TCP) port scanner *strobe.c*
  - PostgreSQL patches
  - etc
- Author of *Cypherpunks: Freedom and the future of the internet*

Jacob Appelbaum

- Independent journalist
- Computer security researcher
  - Core member of Tor
- Hacker
- Representer of Wikileaks in 2010
- Edward Snowden
- Contributed to *Cypherpunks: Freedom and the future of the internet*

Jeremie Zimmermann

- La Quadrature du Net.
- Contributed to *Cypherpunks: Freedom and the future of the internet*

Blackhat vs Greyhat vs Whitehat

- Blackhat
  - Violate security
    - Stealing creditcard numbers
    - Harvest personal data
    - Totally malicious
- Whitehat
  - "Ethical" hackers
  - Use their hacking abilities for good
    - Fix problems blackhat could try to hack
  - Usually they work for a company as penetration testers
- Greyhat
  - Falls between Black/whitehat
  - Usually they don't work for their own
  - May technically commit crimes and do unethical things

Tools we will use

- Linux (Duh!)
- The GNU debugger (GDB)
- Linux debugging tools
  - Strace
  - Ltrace

Section 2

Code debugging

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.
C.A.R. Hoare, The 1980 ACM Turing Award Lecture

When in doubt, use brute force.
Ken Thompson

Companies spend millions of dollars on firewalls, encryption and secure access devices, and its money wasted, because none of these measures address the weakest link in the security chain. *Kevin Mitnick*

Finding bugs

Bugs are annoying, difficult to find and sometimes difficult to fix.

Are there easier ways to debug without using a real debugger?
- Yes, there is!

System calls

In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.[2]

[2] `https://en.wikipedia.org/wiki/System_call`

Strace

Strace monitors the system calls and signals of a specific program. It's helpful when you don't have the source code to properly debug the program.
From the man page;

```
DESCRIPTION
       ....
       strace is a useful diagnostic, instructional,
       and debugging tool. (....)
       (...) Students, hackers and the overly-curious
       will find that a great deal can be learned about
       a system and its system calls by tracing even
       ordinary.(...)
```

Strace examples

```
# trace ls execution
strace /bin/ls
# trace only the open call in ls
strace -e open /bin/ls
# trace mulitple calls in ls
strace -e trace=open,read /bin/ls
# store strace output
strace -o output.txt /bin/ls
# strace a running process (run as root)
strace -e trace=open, read -p /pid/ -o pid.txt
# generate statistic report of system calls
strace -c /bin/ls
```

Strace exercices

See
`http://github.com/security-101/exercices/strace_ex.c`

Ltrace

ltrace intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process.[3]
The different between *ltrace* and *strace* is the relevant information it gives you. Usually, *ltrace* gives a nicer and cleaner output.

[3] http://ltrace.org

Ltrace examples

```
# trace calls to library functions
ltrace /bin/ls
# trace a running process
ltrace -p /PID/
# trace selective library calls
ltrace -e malloc /bin/cat
```

Ltrace exercices

Bugs ≈ vunerabilities

Bugs usually create new vunerability on your code, they should be fixed as soon as possible. It doesn't mean the code runs and it does what it needs to do that there aren't any vunerabilities. This is what we are focussing on *Security 101*; find the vunerabilities, learn how to attack them and then fix them.

Section 3

Dynamic linker tricks

In computing, a dynamic linker is the part of an operating system that loads and links the shared libraries needed by an executable when it is executed (at "run time"), by copying the content of libraries from persistent storage to RAM, and filling jump tables and relocating pointers. The specific operating system and executable format determine how the dynamic linker functions and how it is implemented.[4]

Changing dynamic linkers

A widely used trick, is to change the path of a dynamic linker library, so instead of using the default libraries, the compiler will use your custom linker. For example, when calling `malloc` in a code, it uses the default path to the malloc library, but what happens if we change it's path? What happens if we tell the compiler to link our code with another library?

Normal code	Points to	Library path
code.c	——>	libc.so
code.c	——>	vunerable.so

We could do this without the source code. We just need it's binary. We know that *ls* uses `malloc` (because of *ltrace*), so we could change the dynamic linker to point to a custom `malloc`.

Hacking the code

We use the `LD_PRELOAD` to tell GCC where is the path of our shared objects. That file will be loaded before any other library (including C runtime *libc.so*).
If we compile code.c[5] we will get some output with random numbers. The code uses *rand()* to generate a random number. If we change the `LD_PRELOAD` to load our *vunerable.c* shared object first, we could change the *rand()* function.

```
# compile code.c
gcc -o code code.c
# create our shared object
gcc -shared -fPIC vunerable.c -o unrandom.so
# point to the shared object
LD_PRELOAD=$PWD/unrandom.so ./code
```

[5] see `dynamic_linkers/code.c`

How does it work work?

We told *code.c* to use the real `rand` function, so why did it use the vunerable one?
To check the libraries that *code* loads when executing it's process, we use *ldd*.

```
$ ldd code
  linux-vdso.so.1 (0x00007ffe56bd1000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f385f127000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f385f4c9000)
```

These are the libs needed by *code*. These are built into the executable and determined at compile time. The must be there library is the *libc.so*, this is the files which provides the C functionality; and it also includes `rand`.

How does it work?

By setting LD_PRELOAD, we force some libraries to be loaded for a program.

```
$ LD_PRELOAD=$PWD/unrandom.so ldd a.out
  linux-vdso.so.1 (0x00007ffdc29f1000)
  /home/ephexeve/Documents/org/security-101/
  dynamic_linkers/unrandom.so (0x00007f8fba2ad000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f8fb9f0b000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f8fba4ae000)
```

LD_PRELOAD injection

Using LD_PRELOAD we can inject real code in a way that the application will be able to function normaly. We don't want to break things, if we break things, it will make it harder for us.