Software Development Security

Marc ESPIE

14 mai 2018, promo 2020

Contents

1 BASIC Security **3**
1.1 Introduction . . . 3
1.1.1 Perpectives . . . 3
1.1.2 Coming Exam . . . 3
1.2 Classical companies . . . 3
1.2.1 Classical way companies write specs . . . 3
1.2.2 Auditors . . . 4
1.2.3 Testers . . . 4
1.2.4 Sadly . . . 4
1.3 For a release . . . 4
1.3.1 Mozilla . . . 4
1.3.2 Life of a product . . . 4
1.4 Bugs . . . 4
1.4.1 Security and bugs . . . 4
1.4.2 Finding a bug . . . 5
1.4.3 What to do . . . 5
1.4.4 Little history . . . 5
1.4.5 Reporting a bug . . . 5
1.4.6 Worst case scenario : zero days . . . 5
1.4.7 Bad behaviors . . . 5
1.4.8 Be prepared . . . 5
1.4.9 Closed source . . . 6
1.4.10 How to avoid bugs . . . 6
1.5 Avoiding bug exploit . . . 6
1.5.1 Mitigation . . . 6
1.5.2 Guard pages . . . 6
1.5.3 Better API's . . . 6
1.5.4 The Drepper fallacy . . . 6

2 Unsecure programs **7**
2.1 Buffer Overflow . . . 7
2.2 SQL injection . . . 7
2.3 More about quoting . . . 8
2.4 Printf . . . 8
2.5 Forgotten shells . . . 8
2.6 Services . . . 8
2.7 More sociology . . . 8

3 Trusting code **8**
3.1 Open source . . . 8
3.2 Obstacles . . . 9
3.3 Generated files . . . 9
3.4 Guidelines . . . 9
3.5 Race condition example . . . 9
3.6 Race condition . . . 9
3.7 Correct use of mkstemp . . . 9
3.8 Yet another example . . . 10
3.9 Solutions . . . 10
3.10 ERRNO . . . 10
3.11 Yet another . . . 10
3.12 Beware of hidden global state . . . 10
3.13 Environement . . . 11
3.14 Security in Big Data and AI . . . 11
3.15 mktemp . . . 11
3.16 Error checking is hard. . . . 11
3.17 Portability . . . 11
3.18 He/She said . . . 11
3.19 If you don't have code, you don't have bugs . . . 12
3.20 I'm with stupid . . . 12
3.21 Network . . . 12
3.22 The Unix security model . . . 12
3.22.1 I am root . . . 12
3.23 What rights do I have? . . . 12
3.24 Priv Drop . . . 12
3.25 Too much . . . 12
3.26 Dropping priviledges, the fine print . . . 12
3.27 What about setuid ? . . . 13
3.28 More advice . . . 13
3.29 The root paradox . . . 13
3.30 Designing software . . . 13
3.31 Example: The X window server . . . 13
3.32 fd passing . . . 13
3.33 Auditing code . . . 13

4 Privilege Drop **13**
4.1 Design . . . 14

5 Cross site scripting **14**
5.1 modern . . . 14

1 BASIC Security

One can get the slides at https://www.lse.epita.fr/teaching/courses.html

1.1 Introduction

1.1.1 Perpectives

There are more conferences for attackers than for safety.

Perspectives :
- basic ecosystem of software from security perspective
- vocabulary to pass intership
- Dispell misconceptions about dev security

Setting limits :
- C and Unix
- All the bases of programs are in C or assembly languages

Unix is easier to start with, it's well known and well defined it has existed for more than 20 years. Then it will be not that hard to go on other OSs.

BOOKS:
- Building Secure Software(Viega, McGraw, ISBN 0-201-72152-X)
- Open BSD papers

1.1.2 Coming Exam

Simple questions for the exam:
- You will have access to lecture notes
- So if I ask you to define a term, you shouldn't copy your notes
- You should be able demonstrate that yoou understand the term by explaining it your own words - Give concrete examples
- Create your own examples

Adanced Questions:
- There will be source code and sample audits
- It won't be 100% clean
- It won't be exactly like 'epita standard code'
- If it's different it may not be wrong
- Beware of wrong asssumptions
- The security issues to fix will be nasty ones

Exam in general:
- you can write wether in french or english

1.2 Classical companies

1.2.1 Classical way companies write specs

One person by task

But it's not a good idea.

1.2.2 Auditors

If an auditor finds a bug
...Sometime it's because the design is wrong
Auditors can't catch all
... So devs must know about good practices

You can't have only one developer to remove all the security breaches it's too much.

1.2.3 Testers

You can't have pit testers VS devellopers. A good tester is invaluable.

1.2.4 Sadly

A lot of good databases experts don't even know about SQL injections.

1.3 For a release

1.3.1 Mozilla

When Mozilla ship a new version of firefox, they give the code.
But they have to adapt it for each OS.
So they do modify it before.

So after the realease, they are branchings.
Some people have to still work on correcting bugs.

When working on bugs, you can produce more bugs.

At some point you have to say that some versions are no more supported.

1.3.2 Life of a product

EOL End of life for a product
ESR Extended support release

For Windows XP, microsoft had to support for a very long time this version, because companies needed to use it and didn't wanted to change it.

Companies prefer to have a stable version of a program with security breach a program less stable but with security corrected.

1.4 Bugs

1.4.1 Security and bugs

A bug is not a 'security Hole'.
Most attacks are a serie of bugs.
We want to have defense in depth.
Fixing one bug stops the attack!
An attack is also called an exploit.
Software has vulnerabilities.

1.4.2 Finding a bug

- A developer can find a bug.
- External user find the bug and report it to the company.
- Others can sell this bug.

1.4.3 What to do

Be proactive about a security issue.
Fixing it without letting the bad guys know.

A soon as you find something, you have to fix it, because there will be an soon or later.

1.4.4 Little history

25 years ago, there was a mailing list called bugtraq, but multiple times you had the same new on different OSs.
Now there is CVE
CVE : common vulnerabilities and exposures.

1.4.5 Reporting a bug

Don't do it on Friday
Account for vendors
Have a "secure" channel for bugs

"Meltdown" for Intel processors : is a good example of not what to do.

1.4.6 Worst case scenario : zero days

It's when you see people exploiting a bug.

1.4.7 Bad behaviors

- It's too complicated to be exploitable
It will take time but it will be exploited.

The IE5 url overflow :
Use it to craft assembly code to craft more overflow.
Two month later, a guy used it to write self modifying code.

1.4.8 Be prepared

Sofware components get reused all the time.
Plan to be successfull.

Your code may be used in Hospitals, so nasty guys could exploit it and could kill people.

1.4.9 Closed source

It's no more secure.
lot's of people know how to reverse engineer.

You want to avoid the 'sweep under the carpet effect'

If you do open source, some people will look at what you are doing.

example:
A guy recovered most patterns of buffer overflow from microsoft windows.
It takes one bug...
Everything is exploitable.
You have to carefully check the code that you have been writting.

1.4.10 How to avoid bugs

- Don't do bugs
- Know you APIs
- Prefer secure idioms

1.5 Avoiding bug exploit

1.5.1 Mitigation

In OS, it's an actual technique that will mitigate it.
example : canaries
Function prolog will insert random data on the stack.
This will be checked at the end of the function, if it has changed then, stack smash if called to make program end.

1.5.2 Guard pages

Pages are separed in order to avoid writting on other pages.

1.5.3 Better API's

Don't use strcpy, strcat, strncpy nor strncat.
prefer strlcpy, strlcat; so you can use the size to see if it will fit.

1.5.4 The Drepper fallacy

- 'But I don't write wrong code'
- The reason for slow adoption of strlcpy
For over 10 year Mr Drepper didn't wanted to introduce strlcpy on linux, because people had to write good stuff.
Prefer snprint to sprintf.
When writing code in C, the size have to be obvious.
90% of all software is:
- crap
- unimportant to optimize
- etc...
You can't fix everything
... therefore don't fix everything

This produces "Low-Hanging fruits"
As long as you avoid basic bugs, you're safe from most pirates. (95% of them)

2 Unsecure programs

2.1 Buffer Overflow

Be carefull with malloc overflows.
The programm opening an image can be vulnerable to overflows.
ie: flash with buffer overflows. If you did the right overflow, you could have get more rights than you should have had.
You have to write correctly your library so it is fast and safe.
Beware of undefined behaviors and compiler optimizations.
When working with arrays in C, you want to check the size.
Interger overflow => undefined behavior
You have to check th size of max integer depending on your CPU.

Broken code:

```
int *
alloc_array(int n)
{
  int *t = emalloc(n * sizeof(int));
  return t;
}
int *
read_array()
{
  int s = 0;
  scanf("%d", &s);
  if (s == 0)
  exit(1);
  int *t = alloc_array(s);
  for (int i = 0; i != s; i++)
  scanf("%d", t[i]);
  return t;
}
```

Good practices:
— Use calloc. Make sure that it is safe.
— Maybe craft you own.

2.2 SQL injection

In SQL, never use do.
Don't assume the people you work with know about this.
Never do matching about negative patterns.
ie : Some patterns can be added later, and so you won't be protected anymore.

You should accept the less things you can, and open more things when people ask you to do it. In order to avoid security leaks.

2.3 More about quoting

What you don't know will kill you
Never do matching against negative patterns
e.g., an email address is not something that does not contain some characters
Is is something that only matches a given pattern.
(Subsidiary question: figure out a regexp that matches email)

Do positive matching.

2.4 Printf

What you don't know will kill you.
Printf can print part of memories that you don't want to share.
Beware of executing shell commands from user or asking for paths.

Always write printf("%s", msg);
The compiler will optimize it.

2.5 Forgotten shells

Every time one launches system or popen, a shell is run.

2.6 Services

When a service crash, what do you do ?
-restart service
-don't restart it

The good idea is not to restart it yet, and look at the logs.
So you can know where is the problem.
There may be attackers.

If you have a system that gives you notifications or warning messages, you have to use them.
You should try to attack your owns servers, and make them fail, in order to be sure they fail for the good reasons.
Make sure that you verify regularly that things can fail.
Netflix had some of the same issues, it was client of AWS.
https://www.lemondeinformatique.fr/actualites/lire-netflix-libere-chaos-monkey-dans-la-jungle-open-
html

2.7 More sociology

Usually, the finder of the exploit will let in one or two syntax errors. Script kiddies catch them and fix them and abuse the thing.
They usually do a rm -rf * at the root of the fs...
Keep the logs outside (on a read-only machine) and make backups frequently.
After all, consider how expensive it is to do them and test themand how expensive it is to restart the process.

3 Trusting code

3.1 Open source

If you want people to find bugs in your code, you have to write well documented.
Sometime, people don't look enough. "Many eyeballs fallacy". They found a 20-year-old bug in the OpenBSD libc.

You still have to be really carefull about your stuff.

Code reviews are a good thing. Also make the code readable. Being read by pairs is really gainful.

Trusting people can be dangerous. Being paranoic is a good thing.

One man added a security hole in ssl in python, he was redirecting data to his website. In ssh-decorate. Israeli developer. He detended it wasn't his fault and was hacked. He could also be kidding with ourselves. He removed his software.

How to protect ourselves? Checksums.

Be careful when you do things like wget http://truc.com/bidule.html | bash...

Running code inside a docker is not enough. It can probably access network things, make some shitty processing...

3.2 Obstacles

The systems that give you "just-in-time" tarballs. Make releases and huge version numbers.
... host them elsewhere.

3.3 Generated files

Typically autoconf and automake.
They can be badly generated, and be source of security holes.
If you are using tools that generate code, you have to give the steps to regenerate files.
Never change by hand generated files, it's a very bad practice.
There have already been trojans in the configure files.
It makes it hard to have reliable builds.
We seldom can re-generate the same configure.
Autoconf will compile all it can by default. Do compilation diffs.

3.4 Guidelines

Always make it possible to regenerate everything
... so that people may audit stuff
build should not have access network access.
... and probably have network access.
For instance, in OpenBSD, we switched to doing that, and we caught python/ruby code accessing the network.
... no recent autoconf/automake trojan.

3.5 Race condition example

mktemp example: it does not create the file, only returns a valid file NAME.
Every once in a while, you can have conflicts.
You can overwrite config files, etc.
Use counting.

3.6 Race condition

Trying to access a shared resouce using non-atomic operations.
/tmp is a common directory
mktemp checks the file does not exist
fopen assumes the file still does not exist.
Use mkstemp instead.

3.7 Correct use of mkstemp

* Remove the fd when your code fails.
* errno can change. Save it.

3.8 Yet another example

Config file opener:
Use atomic operations. fstat after fopen.

3.9 Solutions

Know atomic operations.
Prefer fstat, fchmod, fchown to stat, chmod, chown...
ldconfig example: remove the first part. rename will delete the old files.

3.10 ERRNO

You have to be carefull about erno usage. If you do a syscall between a failed syscall and err(), you will report the syscall you did in the middle.

3.11 Yet another

No errno restauration in a signal handler.
printf works with a buffer. Buffer overflow possible.
Careful with memory allocation in signal handlers.
Never use wait (waitpid is ok).

3.12 Beware of hidden global state

— Errno
Use strerror and errc where you can pass them the value we want. if we use errno, we have to save it.

— Locales
If we don't use setlocale, we're in the default "C" locale. Multithreaded programs are more complex (uselocale(3) is a bitch).
Locale affect:
— most isXXXXX functions (encoding). Ex: isalpha.
— printf and scanf
— NOT strcmp
— loaded code.
Faire la distrinction entre les fichiers système et user-facing.
Voir sur localedef pour faire de l'ordre alphabétique...
Chargement dynamique de code aussi: dlopen...

Section critique:
Endroit dans le code où on fixe le comportement.

More generally, try to avoid localized functions when you can.

— blocking status of fd
Signal handlers: Are they set to something non-default?
Does something need them?
Will they create extra errors?
Ex: SIGCHILD: if not caught, can't receive signals.
SIGIGNORE can do stuff.

File descriptors:
SIGPIPE difficult: Default behaviour: program killed.
SIGPIPE -> EPIPE -> EINTR

Do a basic signal handler but do nothing about it.
Comportement process par process obligatoire.
Si signal pendant syscall bloquant, passage par le gestionnaire de signal.
On peut changer ça avec sigaction.

— hidden children

3.13 Environement

It's just an array of strings.

Holds PATH, TERM, TERMCAP. May hold the same variable twice!
PATH: add sbin when the program is admin.
TERM can execute code. Thousands of possible values.
TERMCAP
Gen own environment if we want to be secure.

3.14 Security in Big Data and AI

https://nicholas.carlini.com/

Changes the behaviour of a machine learning algorithm by changing a few bits that aren't perceived by the human eye.
Or emitting inaudible sounds.
Licence plate consequences.

Breaking modern defences against ROP (return-oriented programming).
It's a protection against buffer overflows. He found some instructions that behave as nop.

Evaluation of Chrome Extension security architecture.

3.15 mktemp

BSD and dietlibc return NULL pointer on error.

3.16 Error checking is hard.

Hard to test.

3.17 Portability

Check the results:
— Preprocessor
— unifdef
— nm on .o file

3.18 He/She said

Beware of behavioural differences:
— Nature of char arrays (terminated, not terminated)
— encoding (utf-8, ascii, locale again)
— descriptor vs FILE (NULL vs -1)
— zeroing memory (allocators and OSes)
— empty strings vs NULL pointers.

3.19 If you don't have code, you don't have bugs

— Code that isn't tested is buggy
— ... so don't write code!
— simplify error handling
— don't write code for conditions you can't test
— Group error handling
— Still "fail closed".

3.20 I'm with stupid

free works just fine on NULL pointers. Simplify tests.

3.21 Network

"Be specific in what you receive".

3.22 The Unix security model

In unix, when should you check that you can access a file:
At open: YES
At read/write: NO
At exec: NO (OR YES?!? To be confirmed)
Exception: In tty, revoke() to close all the fds of a program.

3.22.1 I am root

We ignore the rights.
So open the file first, check later (with fstat).

3.23 What rights do I have?

I have the rights of the process
Plus every fd I own.

3.24 Priv Drop

If you need to get some fds reserved to root, start life as root then change your priviledges as soon as you can.

3.25 Too much

closefrom(3), BSD-only.

Solution portable: O_CLOEXE. C'est un flag qui devrait être activé par défaut. Toujours le mettre quand on fait des pipes, des ouvertures de fichiers, etc...

Intéressant de vérifier si le flag est activé par défaut dans les langages de script modernes.

3.26 Dropping priviledges, the fine print

1. Set supplementary groups using setgroups.
2. Set your group id using setgid. **THE ORDER WITH THE NEXT ONE MATTERS !**
3. Set your user id using setuid
4. (You can check your code by inking system("id"))

Beware of Linux:
Make sure you verify setuid/setgid did work (capabilities). This broke sendmail btw)

3.27 What about setuid ?

Effective id: The one that is checked when someone wants to do something.
Real id: What traces you from the beginning.
The man pages from all the setuids, guid, etc. are well written.
Beware of the difference between setuid and set**e**uid.

3.28 More advice

Threading is not all that safe. Secure its accesses.
"Nobody" user: Only for NFS. DO NOT USE IT FOR ANYTHING ELSE. To map the files who belong to a user that does not exist in the client or when the file belonged to root.

3.29 The root paradox

Some programs are safer to run as root. Starting as root allows you to change identity, starting as lambda does not. It's a weakness of UNIX.

3.30 Designing software

A legal obligation if you want to be RGPD-compliant: you can't disappear from a websiteif you don't.
You have to do that from the start.
Work in layers.
* Check that there aren't forbidden characters.
* Then work on finer checks.
This limits noise for audits.

Put up trust barriers.
Ex: A compiler is done in layers.

3.31 Example: The X window server

Graphics cards have DMA access. Huge security breach.

3.32 fd passing

Used for fastCGI (in werb servers).
fds are limited in amount per process. Careful with DDoS attacks.

3.33 Auditing code

Do several passes.

4 Privilege Drop

If a program that opens a file is buggy, and doesn't close a fd, any of its children can access to it.

Add this this to open `O_CLOEXE`

SetUid can change your identity, but you can get back to it at anytime, this should not be used to drop priviledges.
By default the effective code loaded in memory for several program is readomly.

4.1 Design

When designing a system you a the concept of a role.
For example a role could be to receive and parse mails.

You have to decide who can do what.
Trust as less as you can complex code.

5 Cross site scripting

Beware of people inserting `<script> some .js code </script>` in your database.
They might stole your credentials.

2 types of attacks:
— store script in database.
— try to do some request by stealing the cookies of a user.