5. Introduction to Software Security

Computer Security Courses @ POLIMI

Software security fundamentals

Good **software engineering**: meet requirements

- **Functional** requirements
  - Software must do what it is designed for.
- **Non-functional** requirements →IN GENERAL, HARDER TO MANAGE
  - Usability
  - Safety
  - *Security*
- Creating *inherently secure* applications is a fundamental, yet often unknown, skill for a good developer or software engineer.
  - Creating secure software is *hard*.
  - Proof: see next slides.

Software has Vulnerabilities

Software should implement the **specifications**

- Unmet specification == *software bug* IN GENERAL
- Unmet *security* specification == *vulnerability*

A way to leverage a vulnerability to violate the CIA is called *exploit*.

- Vulnerability != exploit.

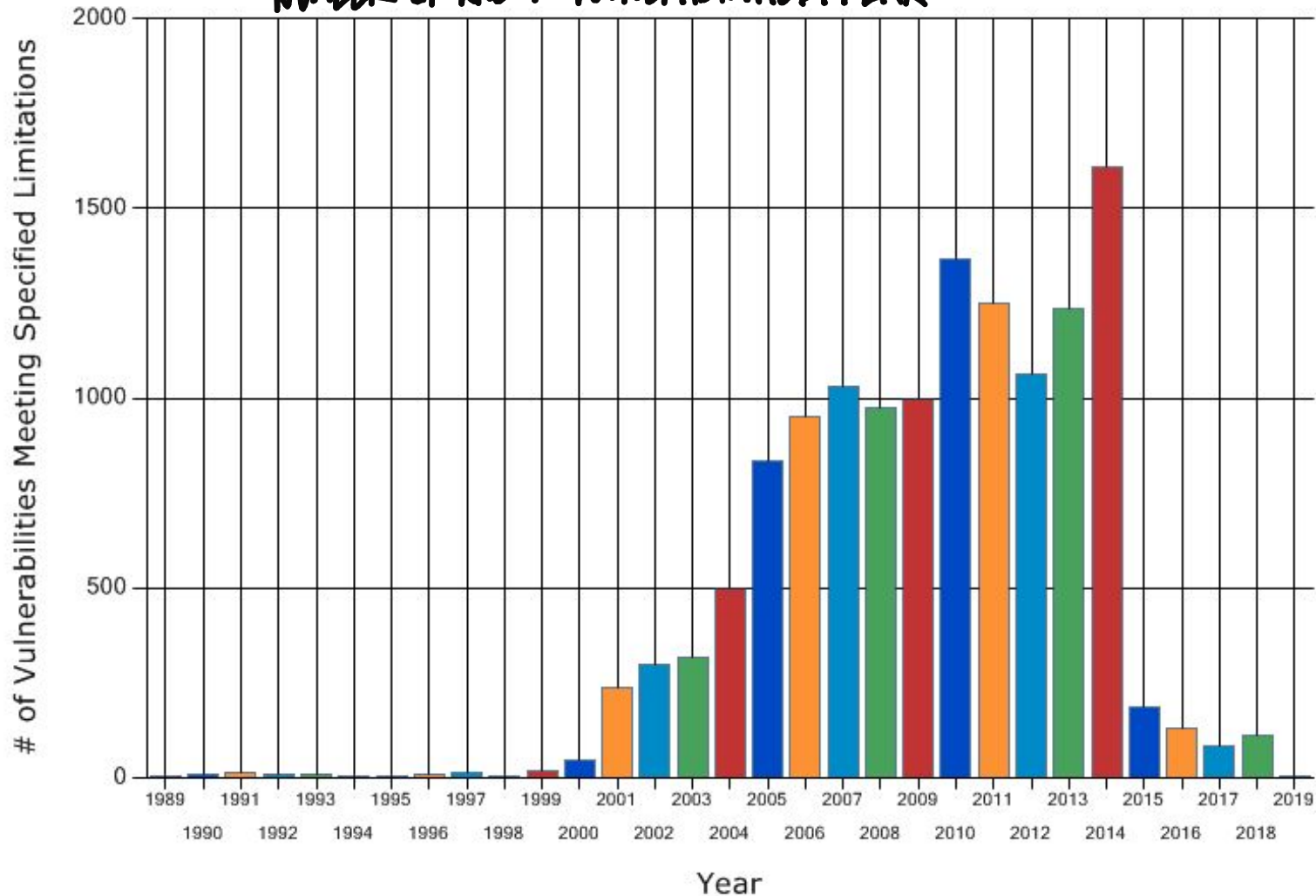AS WE RECALL, IT IS RELATED NOT ONLY TO SOFTWARE BUT ALSO TO PHYSICAL PROPERTIES

PROGRAM THAT INTERACTS WITH THE INPUT OF A VULNERABILITY IN ANOTHER PROGRAM

Life of a SW vulnerability

Known Software Vulnerabilities
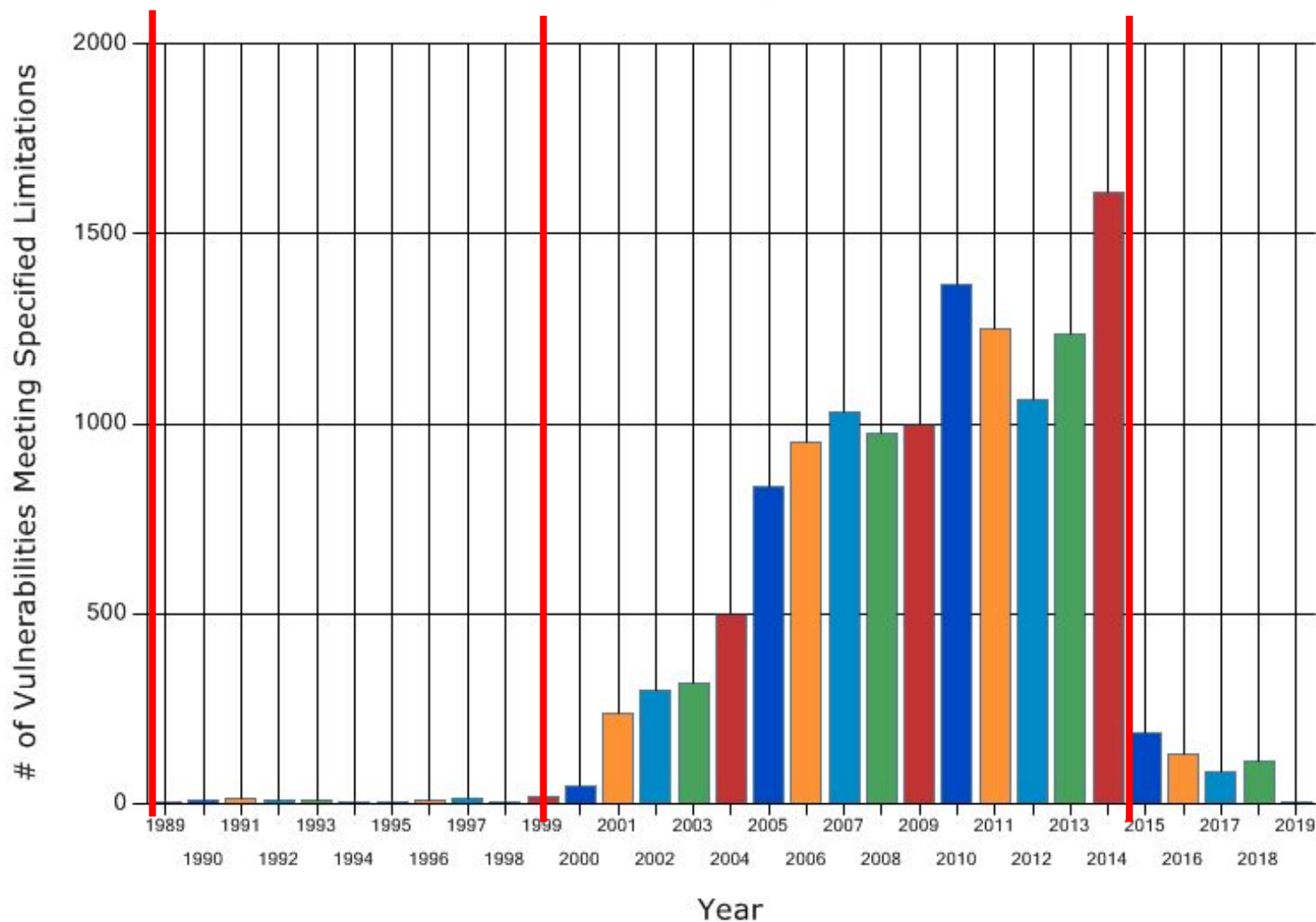

Total Matches By Year

NUMBER OF KNOWN VULNERABILITIES/YEAR

5

Known Software Vulnerabilities

Total Matches By Year

6

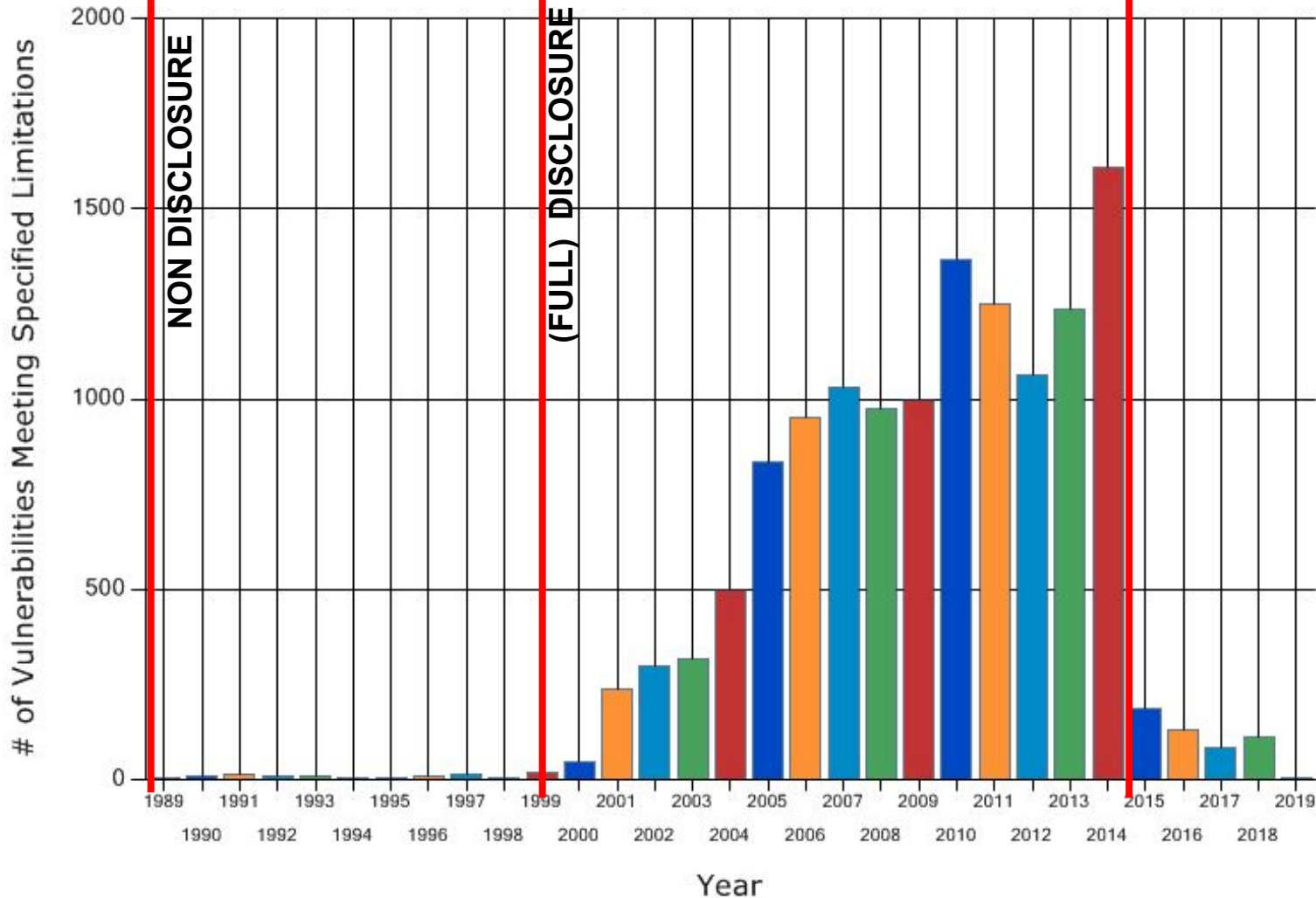# Known Software Vulnerabilities

Total Matches By Year

7
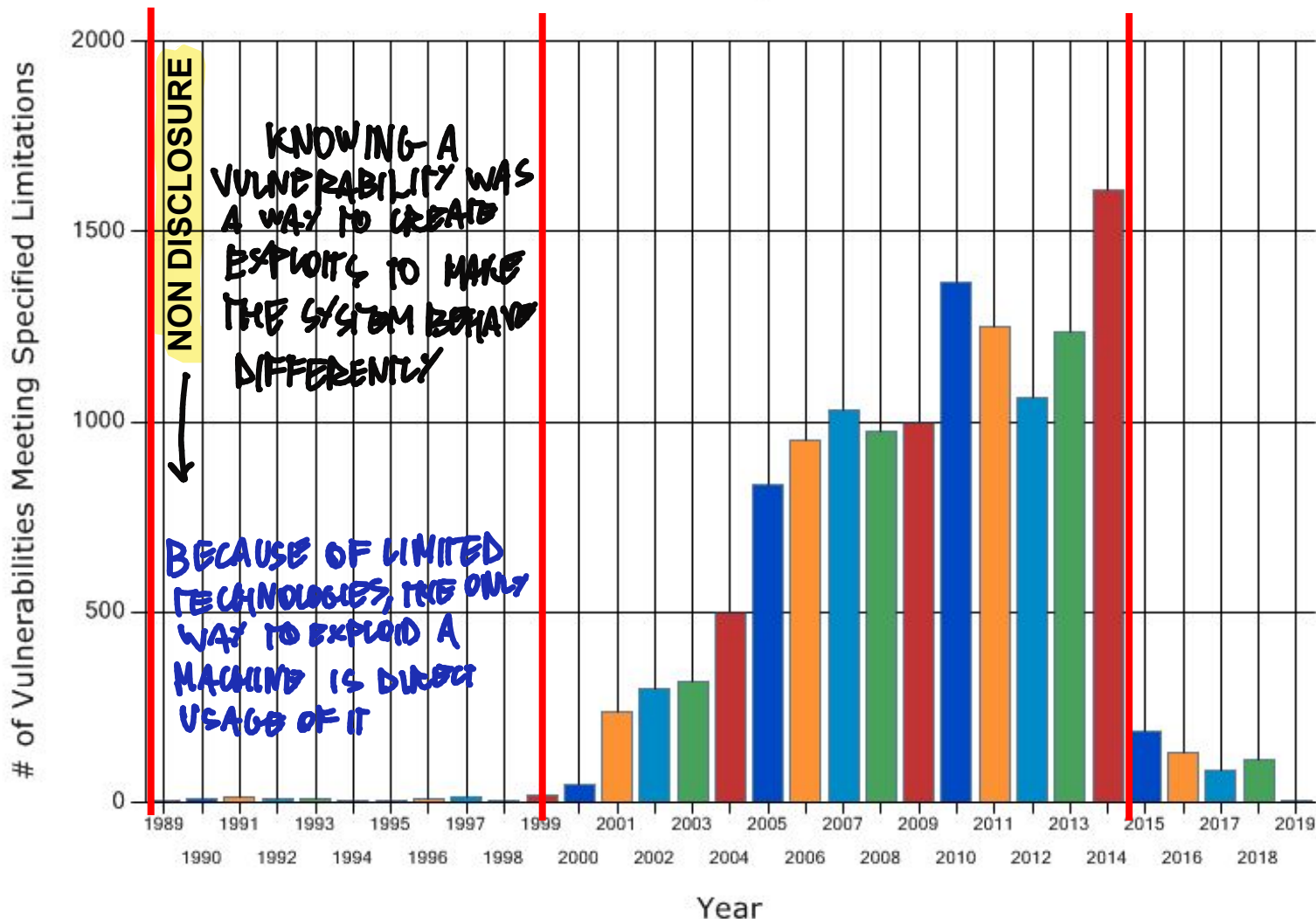
Known Software Vulnerabilities

Total Matches By Year

INTERNET BECAME WIDELY AVAILABLE

NON DISCLOSURE

(FULL) DISCLOSURE

# of Vulnerabilities Meeting Specified Limitations

Year

8

The Early Days of Disclosure

Subject: Comments on the dvwssr.dll vulnerability threads

From: Iván Arce

Date: **2000-04-18** 1:25:52

I do not intend to go further down the full disclosure vs. mediated release of information discussion here, however [Microsoft's handler's] post on NTBugtraq regarding CORE's work requires some clarifications on our side.

[...]

If someone yells 'FIRE' and that appears to be reasonable, I'd would be very careful in my methodology and editorial policies before yelling "NOT TRUE! NOT TRUE! EVERYTHING IS FINE!".
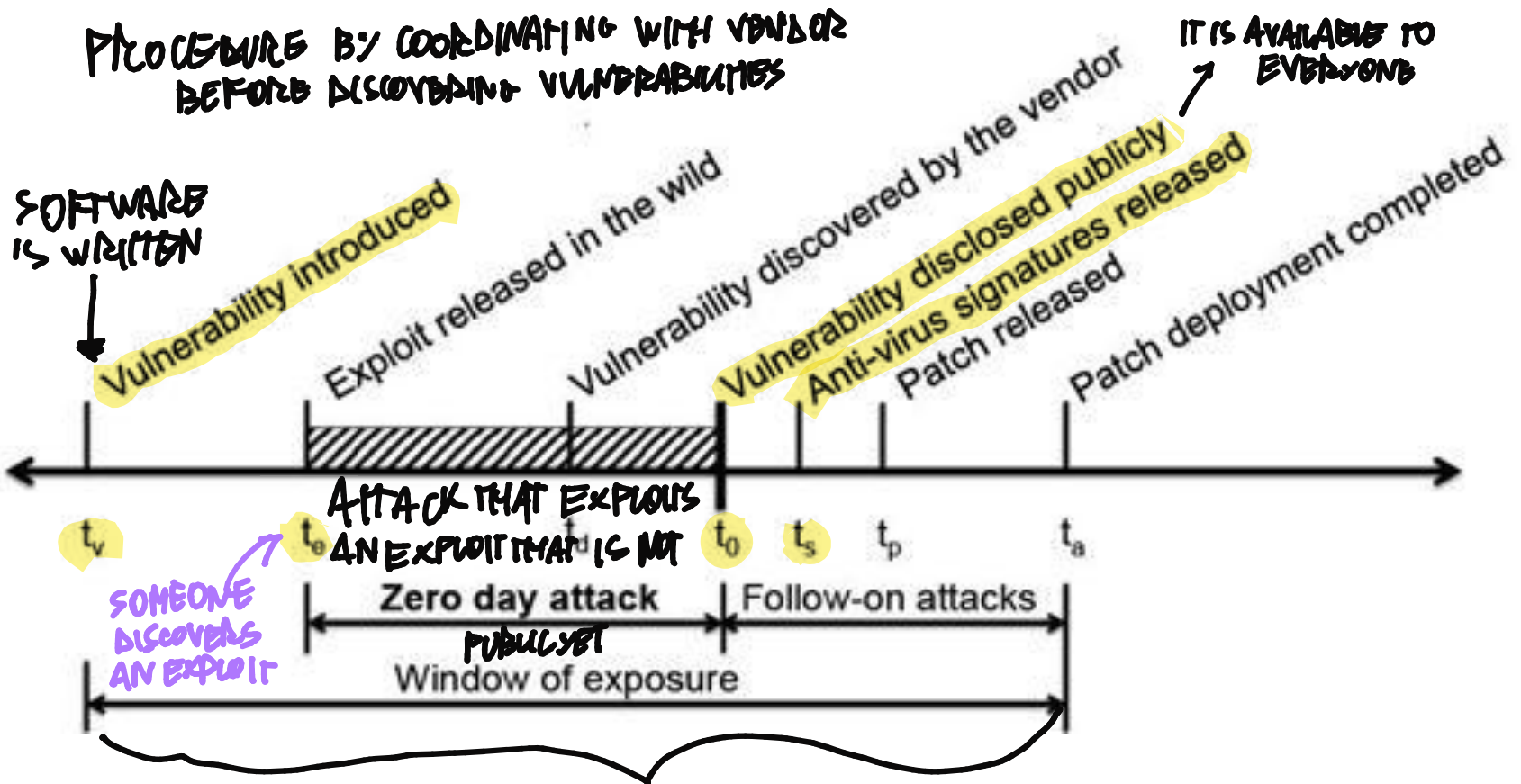
[...]

Excuse me if I'm being rude, but **I'm shocked by the fact that our company is being questioned because we found a bug.**

http://www.s0ftpj.org/bfi/online/bfi10/BFi10-02.html

http://marc.info/?l=vuln-dev&m=95602682515862&w=2

The (full) Disclosure Vuln. Lifecycle

PROCEDURE BY COORDINATING WITH VENDOR
BEFORE DISCOVERING VULNERABILITIES

IT IS AVAILABLE TO EVERYONE

SOFTWARE IS WRITTEN

Vulnerability introduced

Exploit released in the wild

Vulnerability discovered by the vendor

Vulnerability disclosed publicly

Anti-virus signatures released

Patch released

Patch deployment completed

t_v t_e t_0 t_s t_p t_a

ATTACK THAT EXPLOITS AN EXPLOIT THAT IS NOT PUBLICSET

SOMEONE DISCOVERS AN EXPLOIT

Zero day attack   Follow-on attacks

Window of exposure

Leyla Bilge, Tudor Dumitras, Before We Knew It: An Empirical Study of Zero-Day Attacks In The Real World, ACM CCS 2012.
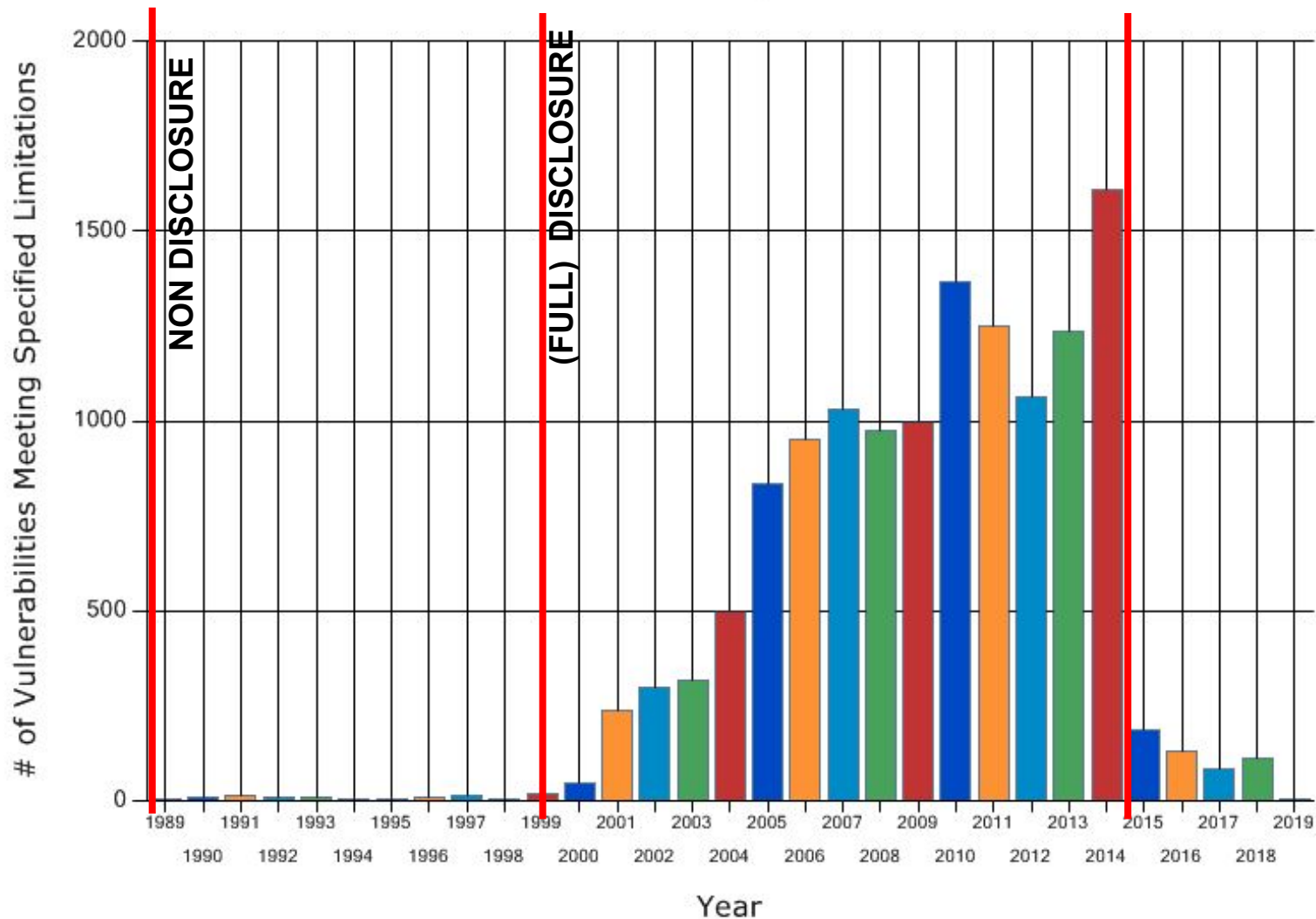
HOW TO REDUCE?   SUPPOSE TO ANTICIPATE t_0

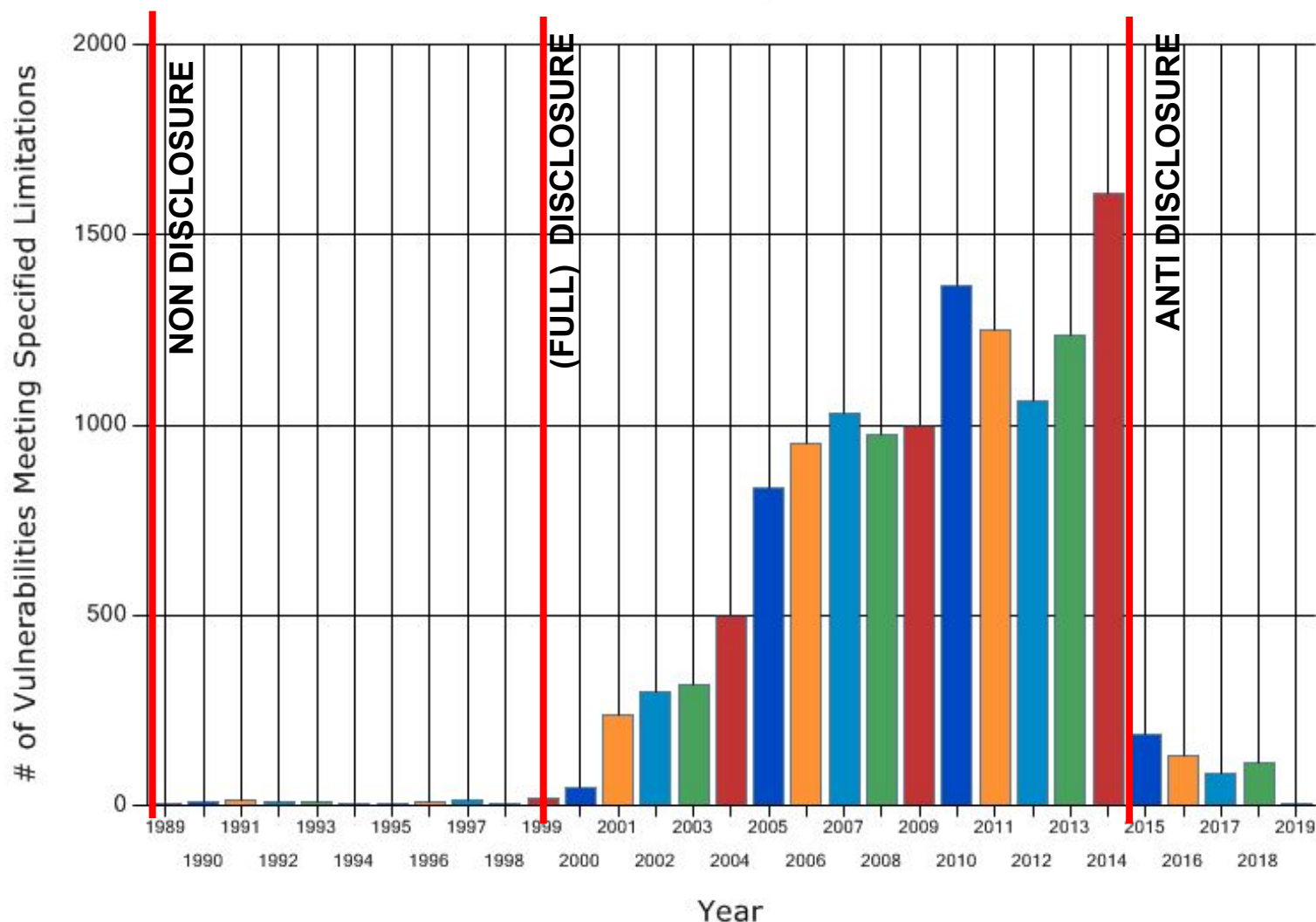# The Black Hat Parties

Known Software Vulnerabilities

Total Matches By Year

12

Known Software Vulnerabilities

Total Matches By Year

13

Black Market of Exploits

Bug Bounties

More Bug Bounties (bugcrowd.com)

Vulnerability and Exploit Example

Processes in UNIX-like systems.

Vulnerability and Exploit Example (1)

Every file has a *owner* (user):

creates → executable

owner:foo

foo
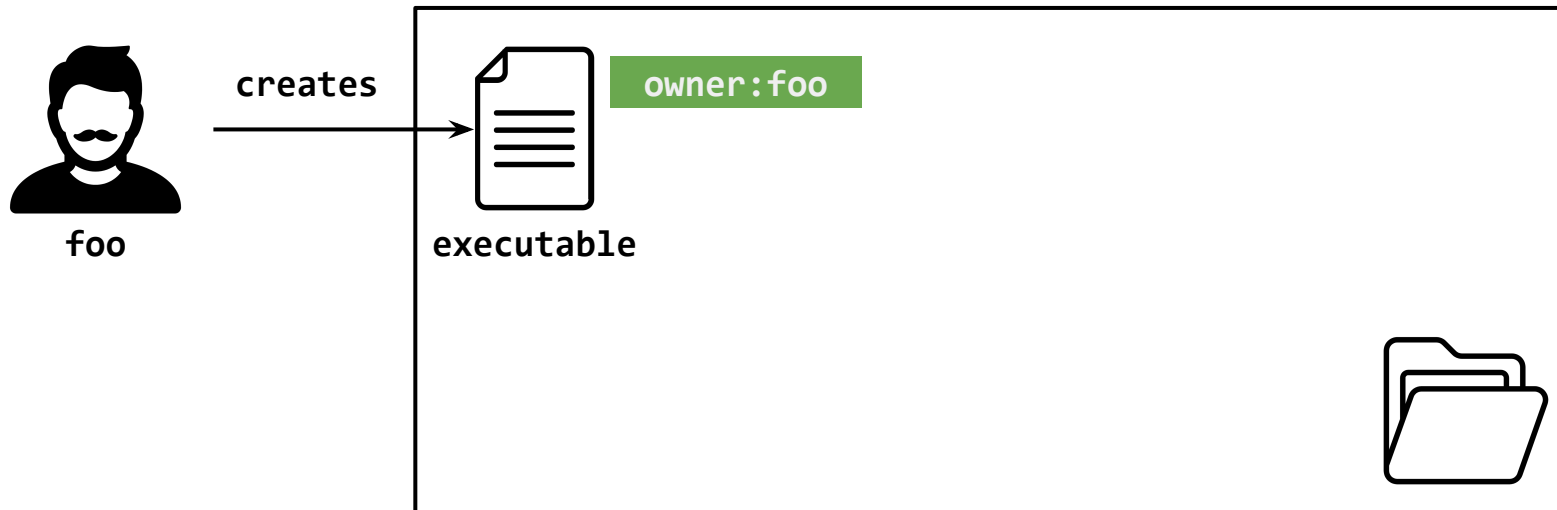
Vulnerability and Exploit Example (1)

"BAR" IS ANOTHER USER THAT CAN CHECK AND USE ALL FILES IN THE DIRECTORY

Every file has a *owner* (user):

```
[bar@localhost]$ ls -la executable
-rwxr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

CAN READ, WRITE AND EXECUTE

creates

owner:foo

foo

executable

ls -la

CAN ONLY READ AND EXECUTE

bar
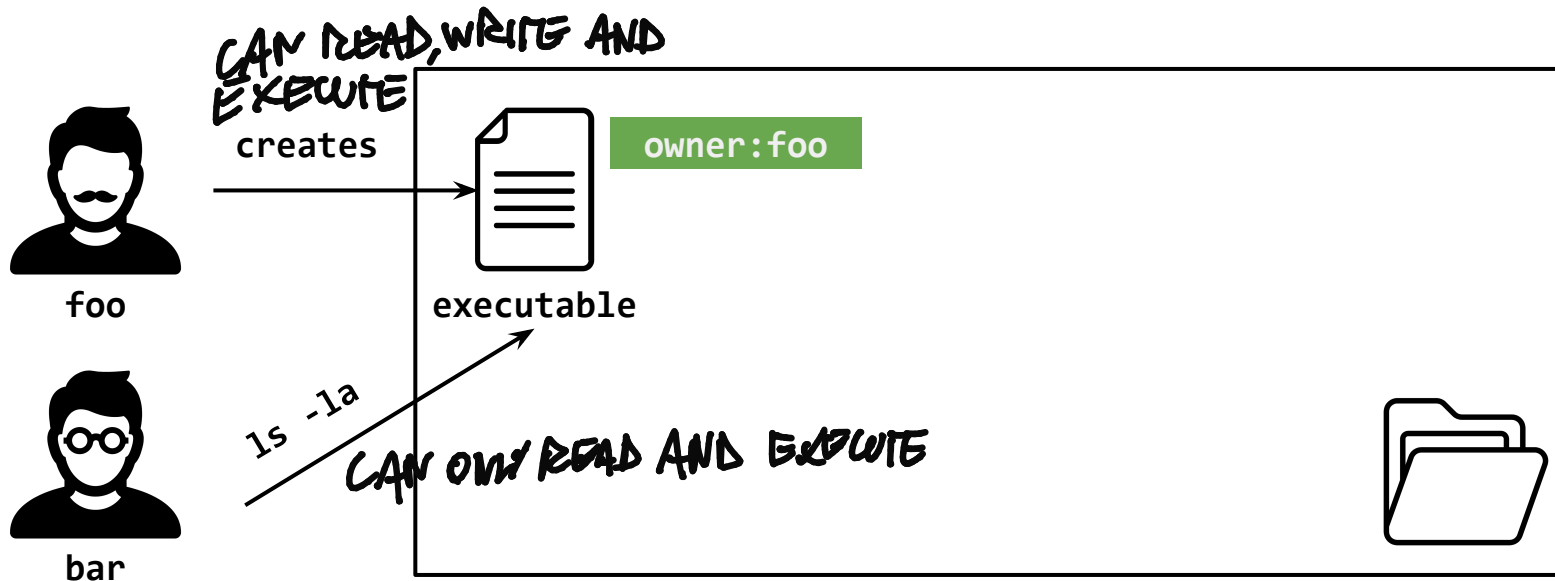
Vulnerability and Exploit Example (1)

WHEN A USER EXECUTES A FILE A NEW PROCESS BEGINS

Every file has a *owner* (user):

```
[bar@localhost]$ ls -la executable
 -rwxr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

Real UID (**RUID**): real *owner* of a process.

```
[bar@localhost]$ ./executable   #Darwin Kernel 13.1.0
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID     COMMAND
 bar    18299  ./executable
```

"REAL USER ID"

The RUID could differ from the owner.

Real UID (RUID): real *owner* of a process
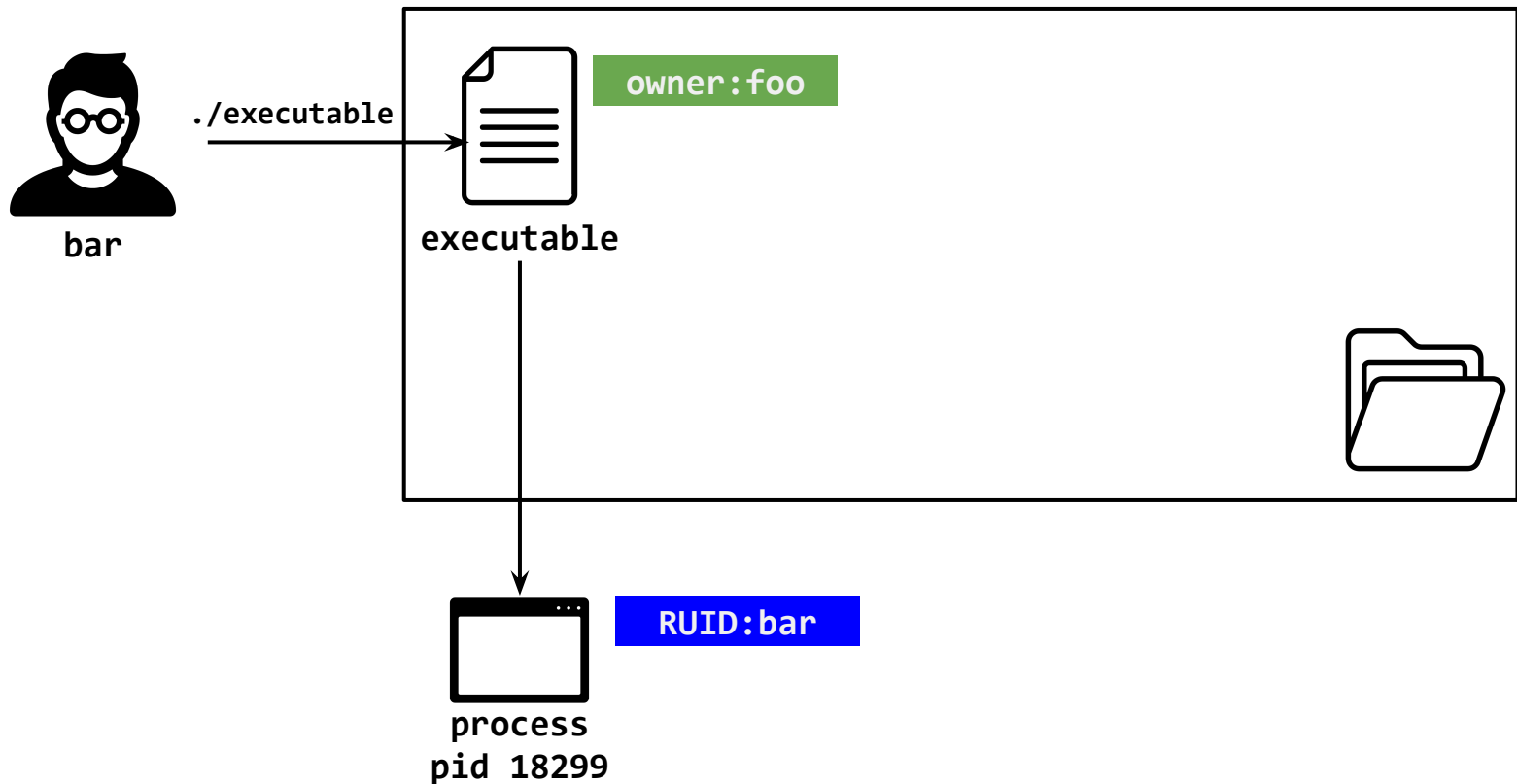
./executable

bar

owner:foo

executable

RUID:bar

process
pid 18299

Vulnerability and Exploit Example (1)

Every file has a *owner* (user):

```
[bar@localhost]$ ls -la executable
 -rwxr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

Real UID (**RUID**): real *owner* of a process.

```
[bar@localhost]$ ./executable   #Darwin Kernel 13.1.0
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID     COMMAND
 bar     18299   ./executable
```

The RUID could differ from the owner.

Effective UID (**EUID**): UID for checking permissions

Vulnerability and Exploit Example (1)

Every file has a *owner* (user):

```
[bar@localhost]$ ls -la executable
 -rwxr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

Real UID (**RUID**): real *owner* of a process.

```
[bar@localhost]$ ./executable   #Darwin Kernel 13.1.0
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID     COMMAND
 bar     18299   ./executable
```

The RUID could differ from the owner.

Normally: RUID == Effective UID (EUID).
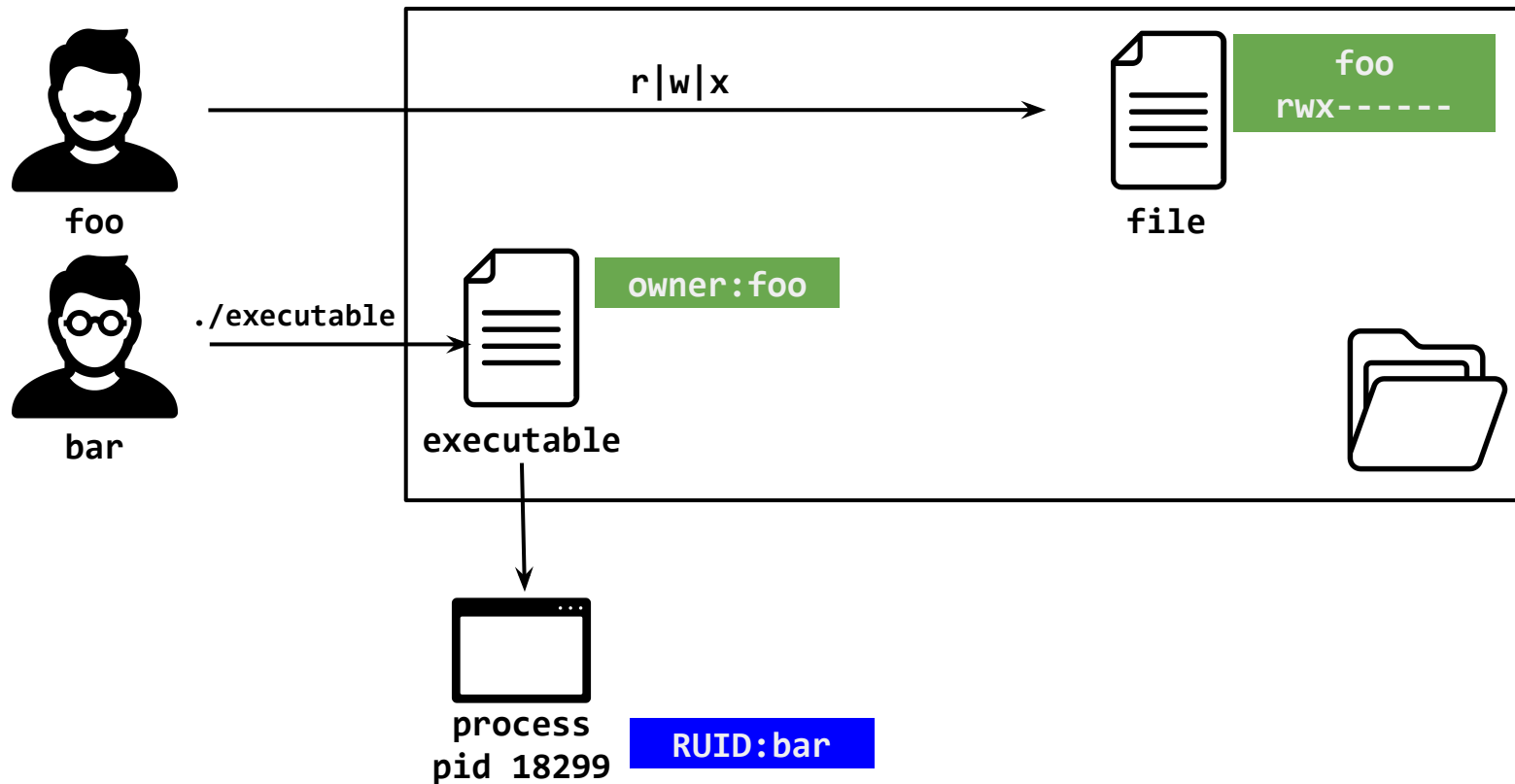
Effective UID (EUID): UID for checking permissions

Effective UID (EUID): UID for checking permissions
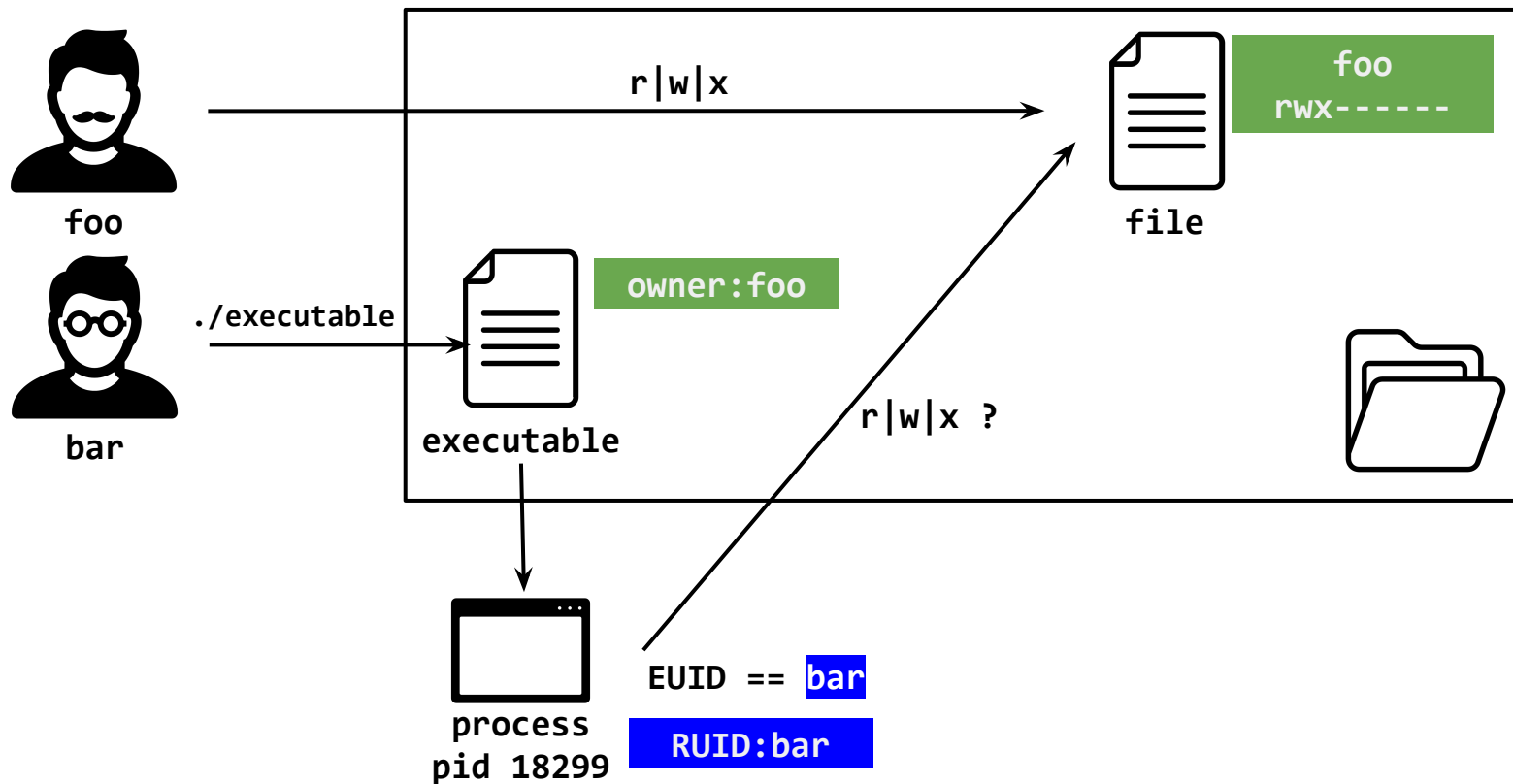
Normally: RUID == Effective UID (EUID).

Effective UID (EUID): UID for checking permissions
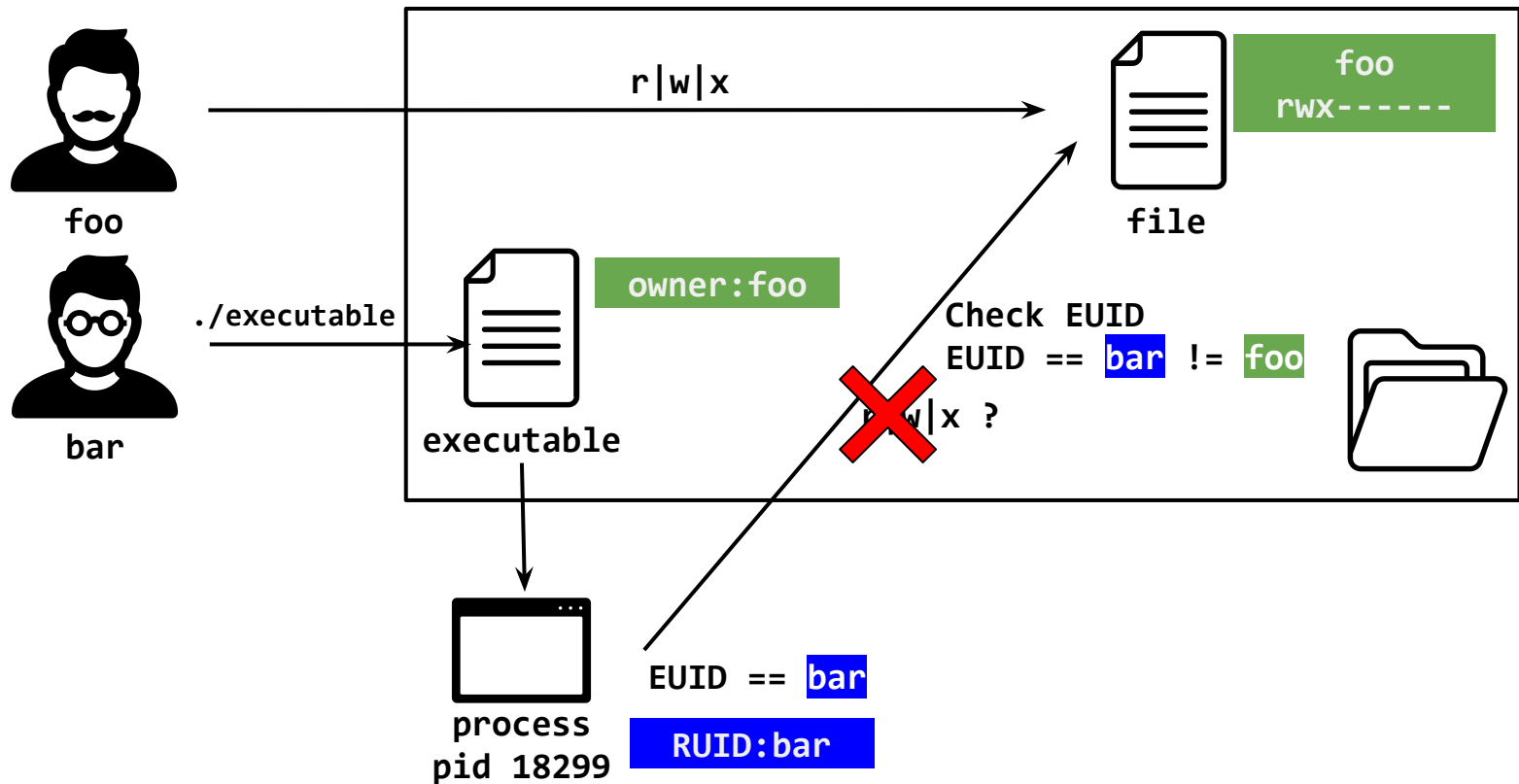
Normally: RUID == Effective UID (EUID).

Vulnerability and Exploit Example (2)

Saved **set-user-ID** (SUID) can be used to change the EUID at runtime.

```
[root@localhost]# chmod u+s executable    CHANGING PERMISSIONS SET
[root@localhost]# ls -la executable       UID TO THE ONE OF THE
 -rwsr-xr-x 1 foo group 41836 2012-10-14 19:19 executable   OWNER OF A FILE
```

Now the executable's SUID is "foo".

```
[bar@localhost]$ ./executable
[bar@localhost]$ ps -a -x -o user,pid,cmd
 USER  PID     COMMAND
 foo    18299   ./executable
```

"bar" == real UID != EUID == "foo".

Now the executable's SUID is "foo".
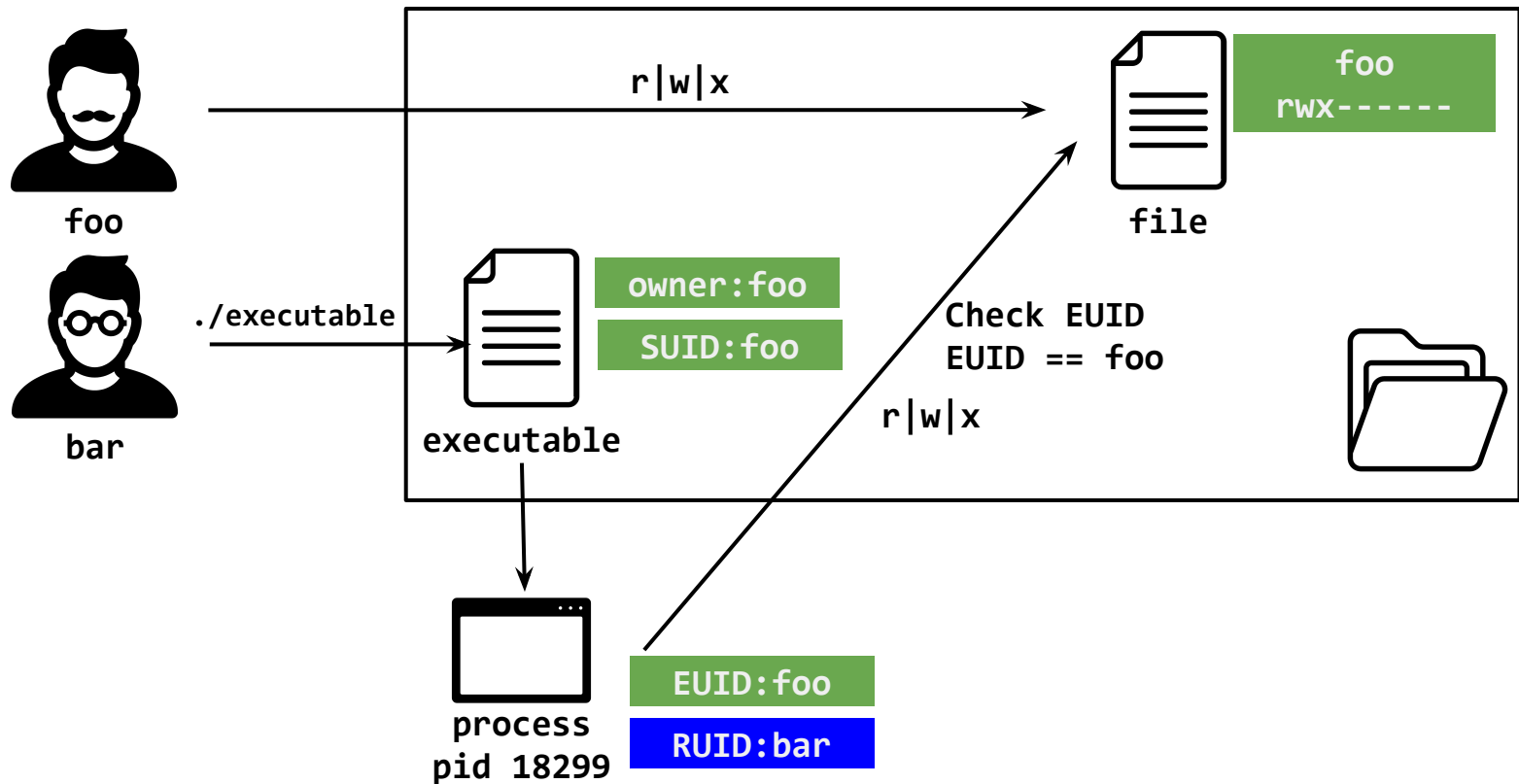
"bar" == real UID != EUID == "foo"

Vulnerability and Exploit Example (3)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    printf("RUID %d EUID %d", getuid(), geteuid());
    return 0;
}
```

Vulnerability and Exploit Example (3)

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    printf("RUID %d EUID %d", getuid(), geteuid());
    return 0;
}
```

```
[foo@localhost]$ gcc -o executable -c executable.c
[foo@localhost]$ sudo su -                    # become root
[root@localhost]# chown root                   # change the owner
[root@localhost]# chmod +s executable      # set the SUID root bit
[root@localhost]# exit                          # get back to foo
[foo@localhost]$ ls -la executable            # check the flags
 -rwsr-xr-x 1 root group 41836 2012-10-14 19:19 executable
```

Vulnerability and Exploit Example (4)

```
[foo@localhost]$ ./executable
RUID 501 Effective 0                    # 501 is foo's UID - 0 is root's UID
```

Vulnerability and Exploit Example (4)

```
[foo@localhost]$ ./executable
RUID 501 Effective 0                          # 501 is foo's UID - 0 is root's UID

[foo@localhost]$ sudo -u root ./executable
RUID 0 Effective 0
```

Vulnerability and Exploit Example (4)

```
[foo@localhost]$ ./executable
RUID 501 Effective 0                        # 501 is foo's UID - 0 is root's UID

[foo@localhost]$ sudo -u root ./executable
RUID 0 Effective 0
```

```
[foo@localhost]$ vim executable.c           // let's add a privileged instruction
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE * fp;
    char line[1024];

    printf("Real %d Effective %d", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");
    while (!feof(fp)) {
        fgets(line, 1024, fp);
        puts(line);
    }
    fclose(fp);
    return 0;
}
```

Vulnerability and Exploit Example (4)

Let's check the permission associated to
`/etc/secret`

```
[foo@localhost]$ ls -la /etc/secret
-rwx------  1 root  wheel  12 Mar 10 16:07 /etc/secret
```

Vulnerability and Exploit Example (4)

```
[foo@localhost]$ ./executable
RUID 501 Effective 0                        # 501 is foo's UID - 0 is root's UID

[foo@localhost]$ sudo -u root ./executable
RUID 0 Effective 0
```

```
[foo@localhost]$ vim executable.c           // let's add a privileged instruction
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE * fp;
    char line[1024];

    printf("Real %d Effective %d", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");          // /etc/secret can be read only by root
    while (!feof(fp)) {
        fgets(line, 1024, fp);
        puts(line);
    }
    fclose(fp);
    return 0;
}
```

Vulnerability and Exploit Example (4)

```
[foo@localhost]$ ls -la /etc/secret
-rwx------  1 root  wheel  12 Mar 10 16:07 /etc/secret

[foo@localhost]$ ./executable

                What happens if we execute the binary file now ?
```

Vulnerability and Exploit Example (4)

Programs are "SUID root" to allow them to execute privileged instructions.

```
[foo@localhost]$ ls -la /etc/secret
-rwx------  1 root  wheel  12 Mar 10 16:07 /etc/secret

[foo@localhost]$ ./executable
Real 501 Effective 0
s3cr3t inf0
```

What if these programs have vulnerabilities or bugs ?

A serious issue....

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    FILE * fp;
    char line[1024];

    printf("Real %d Effective %d", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");         // /etc/secret can be read only by root
    while (!feof(fp)) {
        fgets(line, 1024, fp);
        puts(line);
    }
    fclose(fp);
    return 0;
}
```

Vulnerability and Exploit Example (5)

The EUID should be changed back once the privileged instructions are done.

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
                        // execute as EUID ------------------------

    char line[1024];
    FILE * fp;

    printf("Real %d Effective %d\n", getuid(), geteuid());

    fp = fopen("/etc/secret", "r");
    fgets(line, 1024, fp);
    fclose(fp);

    setuid(501);     //execute as unprivileged user
    printf("Real %d Effective %d\n", getuid(), geteuid());

    puts(line);

    return 0;
}
```

Vulnerability and Exploit Example (5)

```
[foo@localhost]$ ./executable
Real 501 Effective 0
Real 501 Effective 501
s3cr3t inf0                         # <- content of /etc/secret
```

Once we read the file, we release the privileges.

The subsequent instructions are executed with `foo`'s privileges.

General Idea (pseudocode)

<u>Vulnerable program</u>

```
EUID: RUID -> SUID

read(config)

r = parse(config)

IF r = OK do_things() ELSE
error("...")
```

The `read(config)` function prints the content
of the file in the error message.

General Idea (pseudocode)

Vulnerable program

```
EUID: RUID -> SUID

read(config)

r = parse(config)

IF r = OK do_things() ELSE
error("...")


[user@host]$ ./ex /etc/shadow


ERROR in file, line 1:
root:<password hash>: ...
```

The `read(config)` function prints the content of the file in the error message. This allows an unprivileged user to print the content of, e.g., the `/etc/shadow` file, which can be normally read only by privileged users.

General Idea (pseudocode)

Vulnerable program

```
EUID: RUID -> SUID

read(config)

r = parse(config)

IF r = OK do_things() ELSE
error("...")


[user@host]$ ./ex /etc/shadow


ERROR in file, line 1:
root:<password hash>: ...
```

Fixed program

```
EUID: SUID -> RUID

read(config)  //low privs
EUID: RUID -> SUID

r = parse(config)

IF r = OK do_things() ELSE
error("...")

[user@host]$ ./ex /etc/shadow


Permission denied.
```

The `read(config)` function prints the content of the file in the error message. This allows an unprivileged user to print the content of, e.g., the `/etc/shadow` file, which can be normally read only by privileged users.

By acquiring higher privileges only after the file is read, the developer decreases the attack surface and effectively eliminates *this* specific vulnerability (there **may** be *other* vulnerabilities).

44

What else?

Could you spot the other vulnerability in the code snippet on the right?

General Idea (pseudocode)

(still) vulnerable program

```
EUID: SUID -> RUID
read(config)  //low privs
EUID: RUID -> SUID
r = parse(config)
IF r = OK do_things() ELSE
error("...")
```

General Idea (pseudocode)

(still) vulnerable program

```
EUID: SUID -> RUID

read(config)   //low privs

EUID: RUID -> SUID

r = parse(config)

IF r = OK do_things() ELSE
error("...")



[user@host]$ ./ex
carefully-crafted-file
```

Any bug in the `parse(config)` function would
happen in a privileged portion of the code,
therefore potentially allowing the attacker to
perform actions

General Idea (pseudocode)

(still) vulnerable program

```
EUID: SUID -> RUID
read(config)  //low privs
EUID: RUID -> SUID
r = parse(config)
IF r = OK do_things() ELSE
error("...")


[user@host]$ ./ex
carefully-crafted-file
```

Any bug in the `parse(config)` function would happen in a privileged portion of the code, therefore potentially allowing the attacker to perform actions

Fixed program

```
EUID: SUID -> RUID
read(config)  //low privs
r = parse(config)
IF r = OK
    EUID: RUID -> SUID
    do_things()
    EUID: SUID -> RUID
ELSE error("...")
```

By acquiring the privileges as late as possible, and releasing them as soon as possible, the developer decreases further the attack surface (but there may still be *other* vulnerabilities in the "`do_things()`" part of code).

48

Vulnerability *vs.* **Exploit** (Examples)

The developer acquired the privileges before

`read(config)`

Invocation of the program with `/etc/shadow` as the first argument.

The developer acquired the privileges before

`parse(config)`

Invocation of the program on a specifically crafted file to exploit a vulnerability inside the configuration file

Key Issues in Secure Design / Principle of Secure Design (1)

Reduce **privileged** parts to a **minimum**.

KISS (Keep It **Simple**, Stupid). ♥

Discard privileges definitively (i.e. SUID->RUID) as soon as possible

Open design: just as with Kerckhoffs principle, the program must **not rely on obscurity** for security.

Concurrency and race conditions are **tricky**.

Key Issues in Secure Design / Principle of Secure Design (2)

Fail-safe and default deny.

Avoid the use of:
- shared resources (e.g. `mktemp`).
- unknown, untrusted libraries.

Filter the input and the output.

Do not write any own cryptographic primitives, password, and secret management code: use trusted code that has been audited already.

Use trustworthy RNGs such as `/dev/[u]random`

Code Security by Example

We will see 5 main examples of (in)secure programming:

- Memory errors in **desktop applications**
  - Buffer overflow bugs
  - Format string bugs
- Code-injection bugs in **web applications**
  - SQL injection bugs
  - Cross site scripting bugs
  - CSRF

There are many other examples. We will just deal with a few cases.

Conclusion

Bug-free software **does not exist**.

Not all bugs lead to vulnerabilities.

Vulnerability-free software is difficult to achieve.

Vulnerabilities without a working exploit exist.

Be careful with the SUID permission bit.

Material

Section 7.5, 10.6 of D. Gollman, "Computer Security", Wiley (3rd ed.).

["Advanced Linux Programming"](), chapter 10