

5. Introduction to Software Security

Computer Security Courses @ POLIMI
Prof. Carminati & Prof. Zanero

Software security fundamentals

Good **software engineering**: meet requirements

- **Functional** requirements
 - Software must do what it is designed for.
- **Non-functional** requirements
 - 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*
- Unmet *security* specification == *vulnerability*

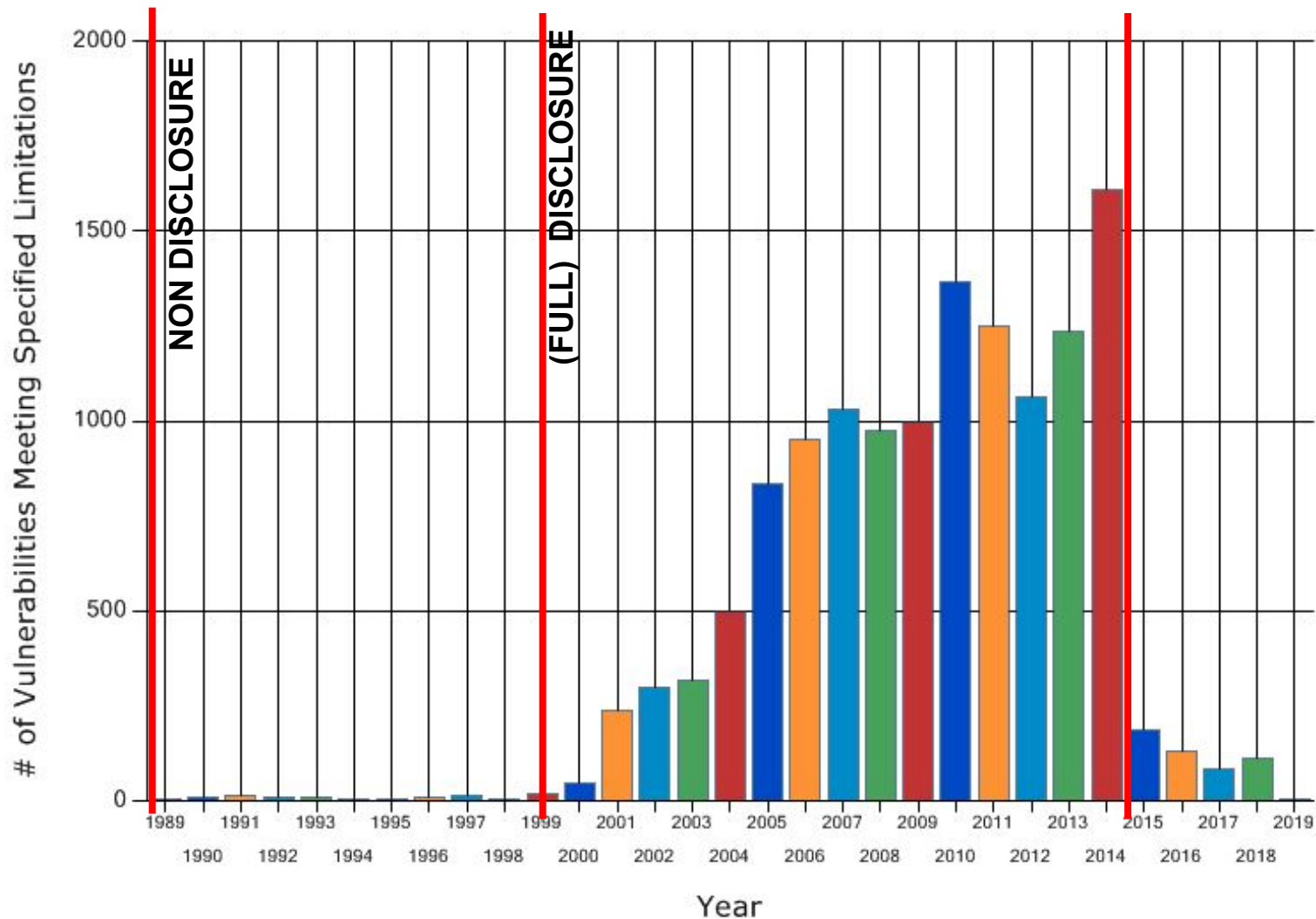
A way to leverage a vulnerability to violate the CIA is called *exploit*.

- Vulnerability != exploit.

Life of a vulnerability

Known Software Vulnerabilities

Total Matches By Year



Source: [NIST' National Vulnerability Database](#)

The Early Days of Disclosure

Subject: Comments on the dwssr.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!".

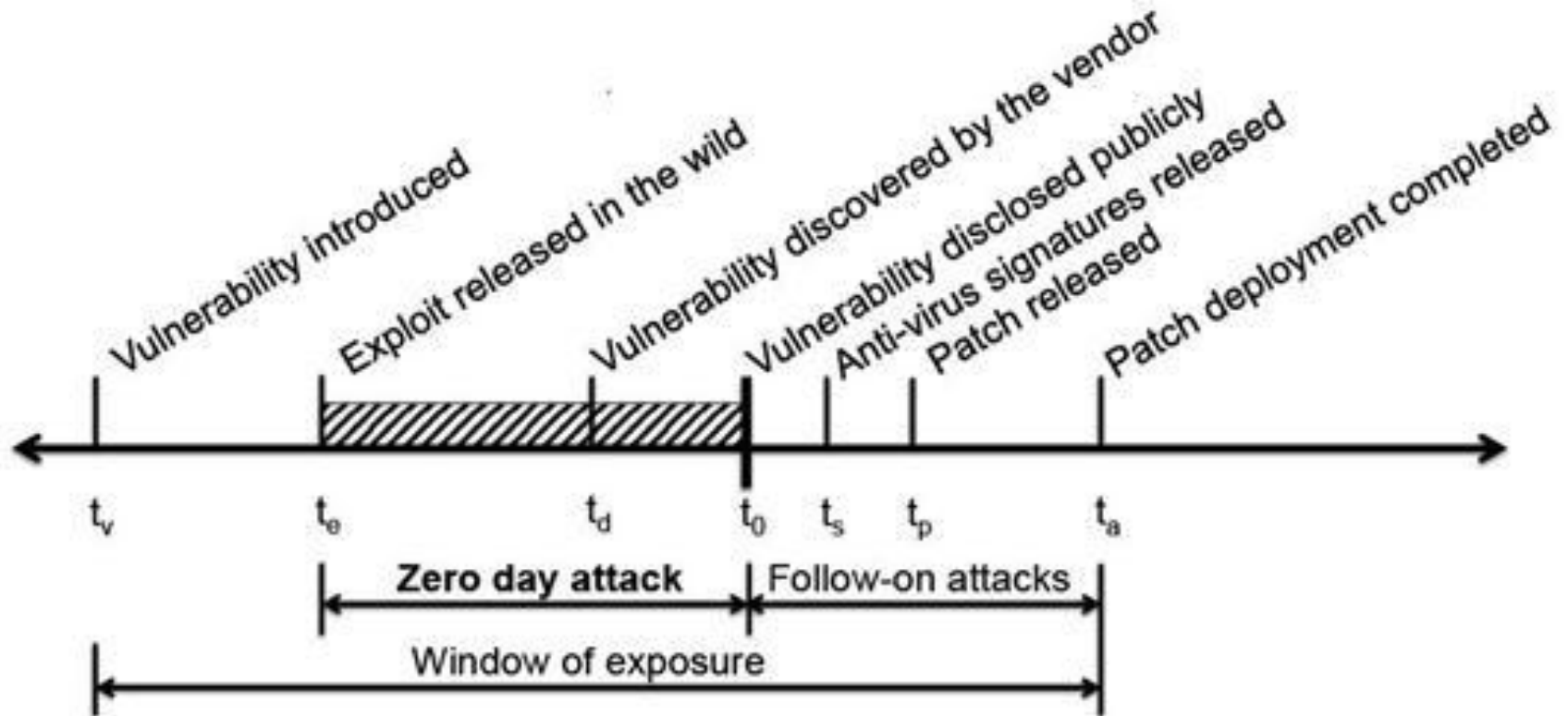
[...]

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



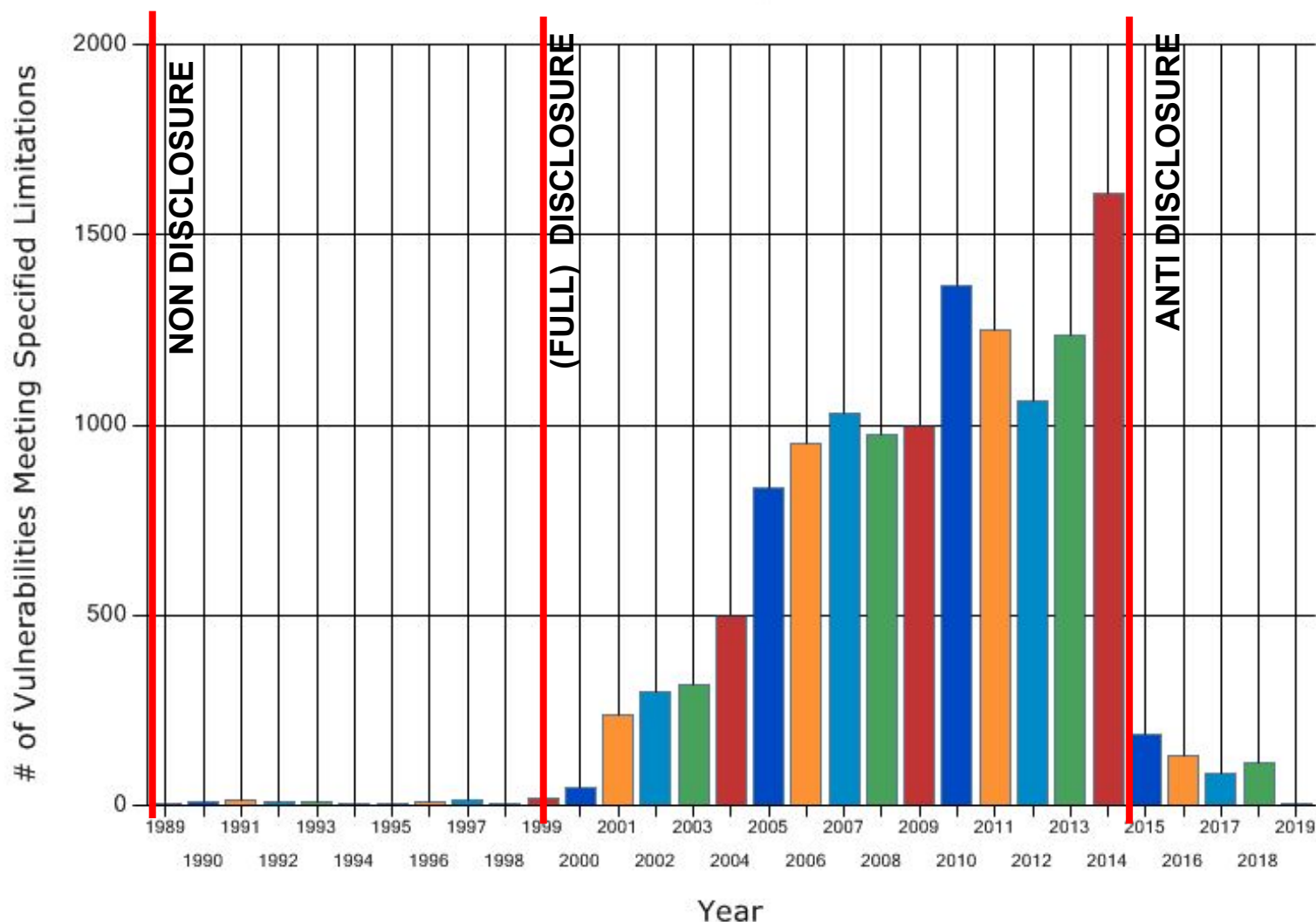
Leyla Bilge, Tudor Dumitras, [Before We Knew It: An Empirical Study of Zero-Day Attacks In The Real World](#), ACM CCS 2012.

The Black Hat Parties



Known Software Vulnerabilities

Total Matches By Year



Source: [NIST' National Vulnerability Database](#)



Black Market of Exploits



Bug Bounties

Build the Next Security Defense Technology and You Could Win \$200,000

WHY ARE WE DOING THIS?

The Microsoft BlueHat Prize contest is designed to generate new ideas for defensive approaches to support computer security. As part of our commitment to a more secure computing experience, we hope to inspire security researchers to develop innovative solutions intended to address serious security threats.

WHAT IS THE CONTEST?

The inaugural Microsoft BlueHat Prize contest challenges security researchers to design a novel runtime mitigation technology designed to prevent the exploitation of memory safety vulnerabilities. The solution considered to be the most innovative by the Microsoft BlueHat Prize board will be presented the grand prize of US \$200,000. Important information:

- Entries will be accepted and must be received by email to bluehatprize@microsoft.com between August 3rd 2011 to midnight Pacific Time on April 1st 2012.
- The winning entry will be announced at Black Hat USA 2012.
- For full details, see rules and regulations.

YOU COULD WIN

First prize: \$200,000 (USD)
Second prize: \$50,000 (USD)
Third prize: MSDN Universal subscription valued at \$10,000 (USD)

QUESTIONS?

Send your questions or comments to bluehatprize@microsoft.com.

EMBED EMAIL SHARE INFORMATION



00:12 / 01:25

HOW DO I ENTER?

To enter, send an email to bluehatprize@microsoft.com — include your technical description and prototype as outlined in the official rules.

The Microsoft BlueHat Prize board will reply with additional information applicants will need to submit a complete entry.

 **SUBMIT EMAIL**

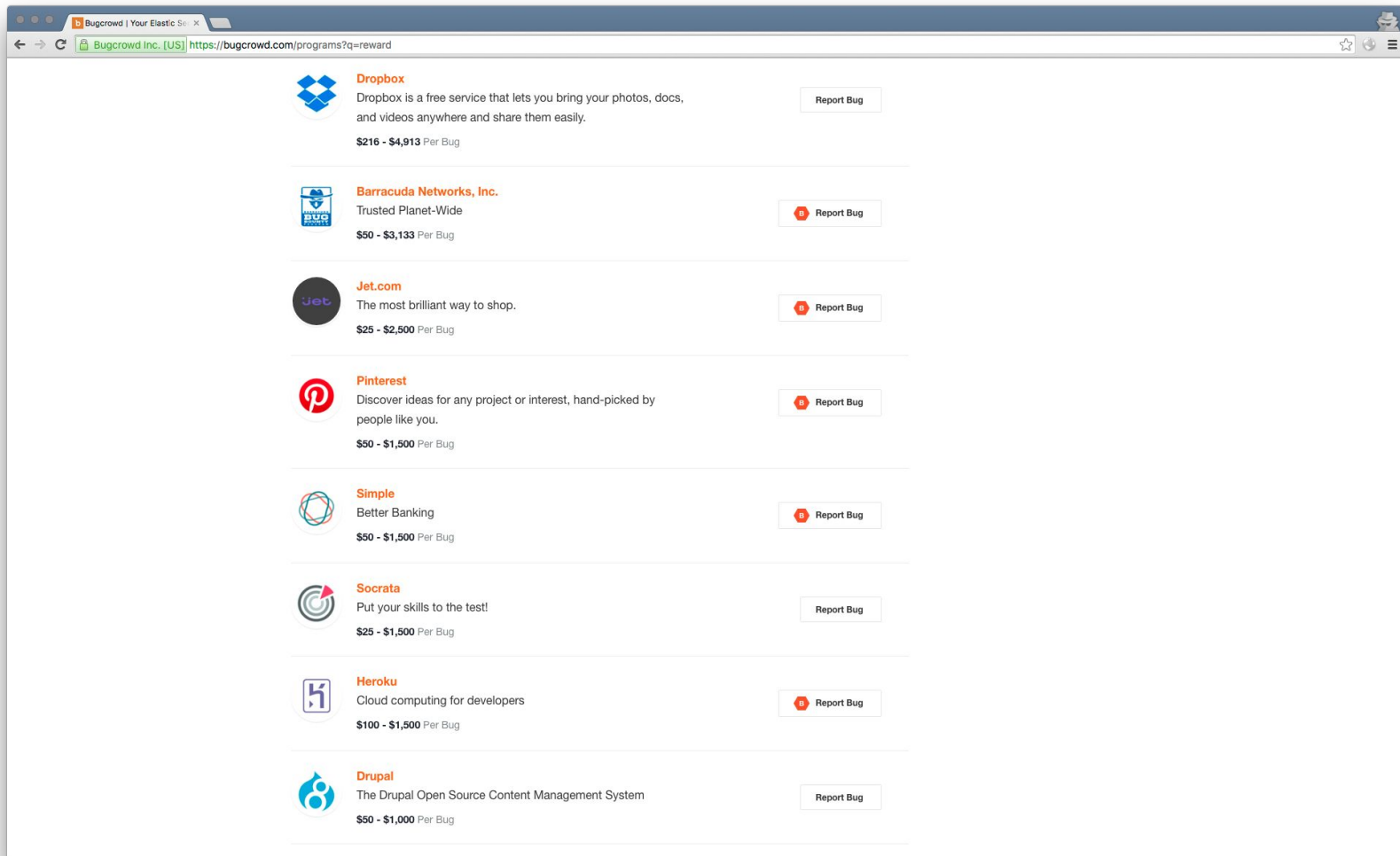
BlueHat Prize

 Tweet 150

 Recommend +43

© 2011 Microsoft Corporation. All Rights Reserved. Terms of Use | Privacy Statement | Trademarks

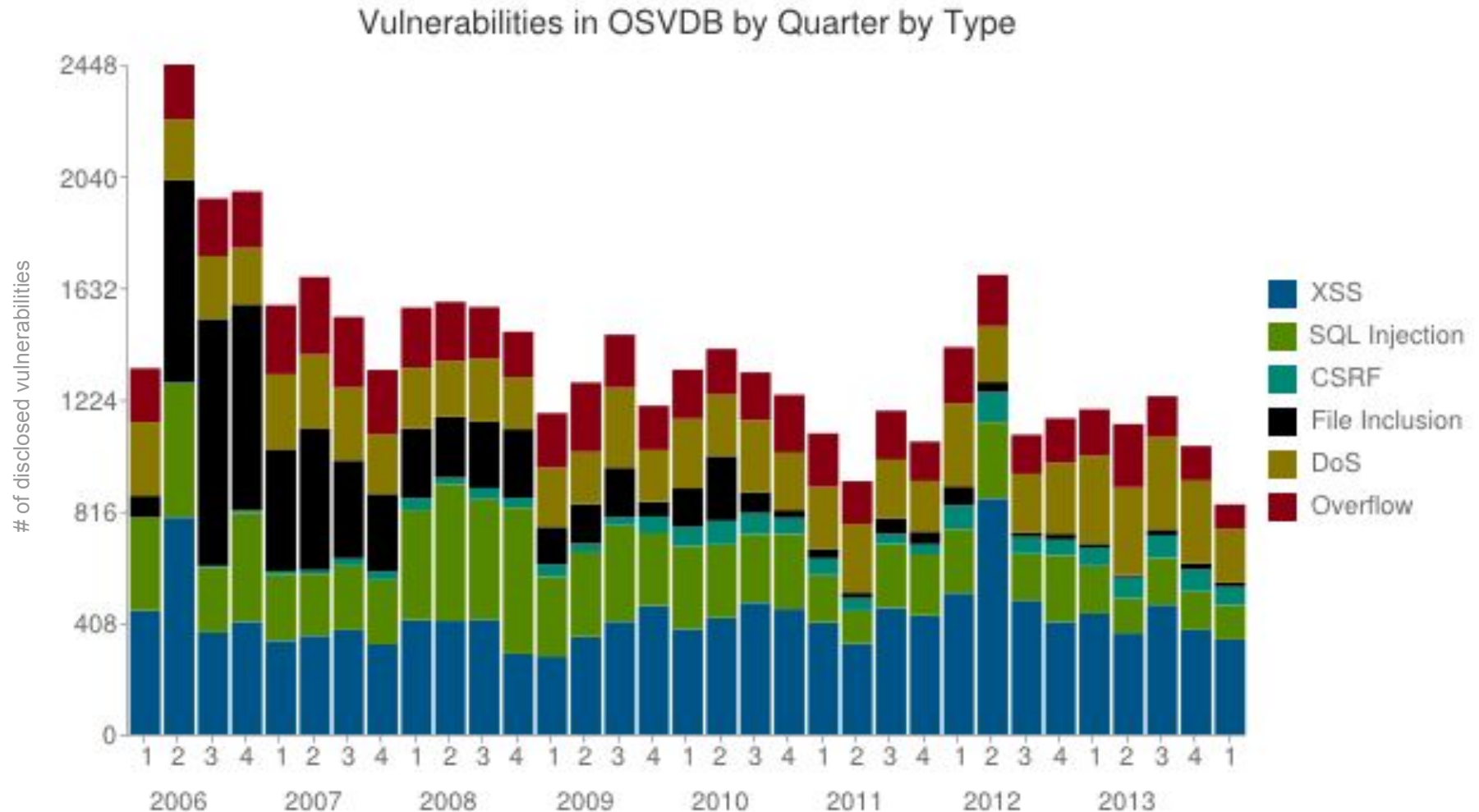
More Bug Bounties (bugcrowd.com)



The screenshot shows the Bugcrowd website with a list of bug bounty programs. The browser address bar shows the URL <https://bugcrowd.com/programs?q=reward>. The list includes programs for Dropbox, Barracuda Networks, Inc., Jet.com, Pinterest, Simple, Socrata, Heroku, and Drupal. Each program entry displays its logo, name, a brief description, the reward range per bug, and a 'Report Bug' button.

Program	Description	Reward Range	Report Bug
Dropbox	Dropbox is a free service that lets you bring your photos, docs, and videos anywhere and share them easily.	\$216 - \$4,913 Per Bug	Report Bug
Barracuda Networks, Inc.	Trusted Planet-Wide	\$50 - \$3,133 Per Bug	Report Bug
Jet.com	The most brilliant way to shop.	\$25 - \$2,500 Per Bug	Report Bug
Pinterest	Discover ideas for any project or interest, hand-picked by people like you.	\$50 - \$1,500 Per Bug	Report Bug
Simple	Better Banking	\$50 - \$1,500 Per Bug	Report Bug
Socrata	Put your skills to the test!	\$25 - \$1,500 Per Bug	Report Bug
Heroku	Cloud computing for developers	\$100 - \$1,500 Per Bug	Report Bug
Drupal	The Drupal Open Source Content Management System	\$50 - \$1,000 Per Bug	Report Bug

Known Software Vulnerabilities (2)



[Source: Open Source Vulnerability Database](#) [CLOSED]

Vulnerability and Exploit Example

Processes in UNIX-like systems.

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).

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
[root@localhost]# ls -la executable
-rwsr-xr-x 1 foo group 41836 2012-10-14 19:19 executable
```

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".

Vulnerability and Exploit Example (3)

```
#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)

```
#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");                // /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)

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
```

Vulnerability and Exploit Example (5)

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

```
#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)

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>: ...
```

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.

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.
```

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).

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("...")
```

```
[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).

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 Kerchoffs 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 crypto, password and secret management code: use trusted code that has been audited already.

Use trusted entropy sources such as `/dev/urandom`

Code Security by Example

We will see 4 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