Software security

Fantastic vulnerabilities and where to find them

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
X/GitHub/...: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

Outline

Introduction

Sometimes, your system is set up/configured correctly, yet a security incident occurs!

IBM found that the average data breach cost 4.88 million in 2024

- This figure includes direct costs, such as fines or lawsuits, as well as indirect costs, such as reputational damage

`https://www.ibm.com/reports/data-breach`

Many breaches involve a human element, but a significant percentage involves the exploitation of vulnerabilities

Terminology

Bug/flaw A coding/design error; basically, defects

Vulnerability A defect that can be exploited, resulting in a negative impact to some security properties (e.g., confidentiality)

Exploitation The act of causing an unintended behavior, by taking advantage of vulnerabilities

0-day/*n*-day A vulnerability that is unknown (0-day) or known for *n* days before its exploitation

Exploit A piece of software (that exploits "something"); typically categorized on how they interact to vulnerable software:

- a remote exploit works over a network and works without any prior access to the vulnerable system
- a local exploit requires prior access to the vulnerable system, and usually increases the privileges

A bit of history

1995 Netscape launched the first bug-bounty program for Netscape Navigator 2.0 Beta, offering cash rewards
http://web.archive.org/web/19970501041756/www101.netscape.com/newsref/pr/newsrelease48.html

1996 Smashing the Stack for Fun and Profit [One96]

- In 2024, "Memory safety vulnerabilities remain a pervasive threat to software security" https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

2001 Open ~~Web~~ Worldwide Application Security Project

- OWASP Top Ten, first published in 2003, aims to raise awareness about application security by identifying the most critical risks

2004 (2008, publicly) Microsoft Security Development Lifecycle
https://www.microsoft.com/en-us/securityengineering/sdl/

Vulnerability classification: CVE vs CWE

- Specific instances, within a product or system, of known vulnerabilities are collected as CVE, Common Vulnerabilities and Exposures — `https://cve.org/`
  - E.g., *CVE-2020-9641*
    Adobe Illustrator v24.1.2 and earlier have a memory corruption vulnerability. Exploitation could lead to arbitrary code execution.
  - This site does *not* provide proof of concepts/exploits; a repository for exploits and PoC, rather than advisories, is `https://www.exploit-db.com`
- Common classes of vulnerabilities, not specific to products/systems, are collected as CWE, Common Weakness Enumeration — `https://cwe.mitre.org/`
  - E.g., *CWE-242: Use of Inherently Dangerous Function*
    Program calls a function that can never be guaranteed to work safely.

A good, yet dated, book is *24 Deadly Sins of Software Security* [HLV10]

Languages

> *Software with security vulnerabilities can be written in any programming language. Still, the programming language can make a difference here, by the language features it provides (or omits), and by the programmer support it offers for these, in the form of compilers, type checkers, run-time execution engines [Pol17]*

For instance, C and C++ have a lot of *undefined* behaviours [WCC+12]
E.g., what does it happen when you divide an `int` by `0`?

A very interesting talk is "Garbage In, Garbage Out: Arguing about Undefined Behavior" by Chandler Carruth @ CppCon 2016 `https://youtu.be/yG1OZ69H_-o`
For more, see [DMS06]

Programming languages (updated: October 2024)

Which are the most popular/used today? (not necessarily the best)

- Python
- C/C++
- Java/Visual Basic.NET/C#
- PHP/Javascript/R
- Assembly in 16^{th} position (!)

See, for instance:

- `https://www.tiobe.com/tiobe-index/` ... popularity ... based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines ... *is not about the best* ...
- `https://spectrum.ieee.org/top-programming-languages-2024`

A common misconception

As Michael Howard, a program manager on the Microsoft Security Engineering Team, says [LH02],

Security features != Secure features

For a program to be secure, all its portions must be secure, not just the parts that explicitly address security (e.g., access control mechanisms)

- security features are typically implemented with the idea that they must function correctly to maintain system security
- non-security features are often the ones that can go wrong and lead to security problems
- like a chain, a program is only as strong as its weakest link

Correctness != Security

(non-security) Testing is not enough:

- Testing typically means running down the list of requirements, making sure they are fulfilled
- If the software fails to meet a particular requirement $\rightarrow$ bug
- Security problems are frequently "unintended functionalities", not requirement violations

quoting Ivan Arce, CTO of Core Security Technologies

> *Reliable software does what it is supposed to do. Secure software does what it is supposed to do, and nothing else.*

Attacker controlled input

If you assume the existence of adversaries, who intentionally try to subvert the system, all inputs are potentially malicious

- you must validate and/or sanitize all inputs
- the sooner, the better: data flows through the system, and the more it flows, the more difficult it is to track and validate

Taint Analysis

Taint analysis follows the flow of data through a program, from sources, i.e., attacker-controlled inputs, to sinks, i.e., possibly dangerous functions. When the value of a variable depends on a source, it is considered tainted. The idea is to analyze all paths from sources to sinks, to determine if tainted data can reach a sink.

Ingredients for secure coding

- Perform regular code reviews
  - This requires knowledge; practitioners with little experience: do not know what to look for, or what to do about problems
    - A good reference on security assessment is [DMS06]
    - An effective way to improve is studying past errors to prevent them from happening again
- Leverage static analysis tools [CW07] to
  - quickly check many possibilities and corner cases
  - explore a large number of "what if" scenarios, *without* running your (incomplete?) code
- Discover remaining vulnerabilities dynamically, by leveraging:
  - fuzzers
  - sanitizers
  - ...

Fixed-size binary integers

Integers are usually represented as fixed-size binary numbers; e.g.,

Three-bit integers

Bits	Unsigned value	Signed value (Two's complement)
000	0	0
001	1	1
010	2	2
011	3	3
100	4	−4
101	5	−3
110	6	−2
111	7	−1

`https://en.wikipedia.org/wiki/Two%27s_complement`

Integers in programming languages

What many programming languages call *integers* are not $\mathbb{Z}$

- they are $\mathbb{Z}_{2^n}$, the ring of integers modulo 2^n, where n is the number of bits used to represent those values
- we write, e.g., 42 meaning [42], the equivalence class of 42 modulo 2^n

Quick quiz

Suppose to use just 3 bits, then $n = 3$ and $2^n = 8$

- In $\mathbb{Z}_8$, $2+3=$
- In $\mathbb{Z}_8$, $6+3=$
- Can you find $x, y \in \mathbb{Z}_8 \setminus \{0\}$ s.t. $x \cdot y = 0$?

Moreover, $-1 = 7$, $-2 = 6$, and so on.

Overflow and Underflow

Typically, n bits are used to represent either the interval of

- unsigned "integers": $[0, 2^n - 1]$ or
- signed "integers": $[-2^{n-1}, 2^{n-1} - 1]$

An integer overflow occurs when a result cannot be represented, because it is higher/lower than the maximum/minimum representable value

- when not hadled, overflows can compromise reliability and security
- some texts use *underflow* when the result is lower than the minimum

Commonly, overflow and underflow results wrap around

Truncation and extension

In C/C++, what happens when the assignment `a=b` is executed? Or when the expression `a+b` is evaluated?

It depends on:

- `sizeof(a)` and `sizeof(b)`
- the "signedness" of `a` and `b`

...and can be rather tricky: generally, mixing signed and unsigned yield unsigned.

When `sizeof(a)!=sizeof(b)` there is truncation or 0/sign-extension

In [DMS06] you can find about 100 pages on C issues!

Truncation/extension example

```
#include <stdio.h>
int main()
{
    int i = 0xcafebabe; /* in this example sizeof(int) is 4 */
    short s = i;
    unsigned short us = i;
    signed char c = i;
    unsigned char uc = i;
    printf("i=%x\n", i);
    printf("s=%x us=%x\n", (int)s, (int)us);
    printf("c=%x uc=%x\n", (int)c, (int)uc); }
```

the result is:

```
i=cafebabe
s=ffffbabe us=babe
c=ffffffbe uc=be
```

From the C++11 standard [ANS12]

Section 3.9.1

Unsigned integers . . . shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.

A footnote clarifies that: "This implies that unsigned arithmetic does not overflow . . . "

Section 5 [expr]

If . . . the result is . . . not in the range of representable values for its type, the behavior is undefined

Bound checking (?)

```c
if (scanf("%d %d", &i, &j)!=2) { /* ... */ return; } /*...*/
if (i<0 || j<0) {
        printf("i and j must be non-negative!\n");
        return; }
int k = i + j;
if (k >= 100) {
        printf("i and j are too big!\n");
        return; }
printf("%d is in the interval [0, 99]\n", k);
/* (***) use k; e.g. to index an array of 100 elements */
```

Quick quiz

Can you find `i` and `j` s.t. `f(i, j)` reaches `(***)` but `k` $\notin [0, 99]$?

Arithmetic checking (?)

```
int f(int a, int b, int c) {
        if (b==0 || c==0) {
                printf("b and c cannot be zero!\n");
                return 0;
        }
        return a / (b*c);
}
```

Quick quiz

Can you find three arguments that make `f` perform a division by zero?

Allocation checking (?)

```
int *array_copy(int *array, int len) {
        if (array==0 || len<0) return 0;
        int *copy, i;
        copy = malloc(len * sizeof(int));
        if (copy == 0)
            return 0;
        for(i = 0; i < len; i++)
            copy[i] = array[i];
        return copy
}
```

Quick quiz

What can go wrong here?

Real-world example 1: OpenSSH 2.3.1–3.3 (2002)

> *Serveral versions of OpenSSH's sshd between 2.3.1 and 3.3 contain an input validation error that can result in an integer overflow and privilege escalation.*

- `https://www.kb.cert.org/vuls/id/369347`
- `https://www.openssh.com/txt/preauth.adv`

Real-world example 2: Stagefright (July, 2015)

Stagefright is the collective name for a group of software bugs that affect Android operating system, allowing an attacker to perform arbitrary operations on the victim device through remote code execution and privilege escalation.

Security researchers demonstrate the bugs with a proof of concept that sends specially crafted MMS messages to the victim device and in most cases requires no end-user actions upon message reception to succeed, while using the phone number as the only target information.

The underlying attack vector exploits certain integer overflow vulnerabilities

`https://www.kb.cert.org/vuls/id/924951`

Real world example 3: Boeing 787 (2020)

The US Federal Aviation Administration has ordered Boeing 787 operators to switch their aircraft off and on every 51 days to prevent what it called "several potentially catastrophic failure scenarios" — including the crashing of onboard network switches. ...
According to the directive itself, if the aircraft is powered on for more than 51 days this can lead to "display of misleading data" to the pilots, with that data including airspeed, attitude, altitude and engine operating indications. On top of all that, the stall warning horn and overspeed horn also stop working. ...
The problem? They put a millisecond clock with a 32-bit register and it overflows.

- https://twitter.com/mountain_ghosts/status/1245754158910705668
- www.theregister.co.uk/2020/04/02/boeing_787_power_cycle_51_days_stale_data/

Real world example 4: Sequoia (2021)

We discovered a size_t-to-int conversion vulnerability in the Linux kernel's filesystem layer: . . .
We successfully exploited this uncontrolled out-of-bounds write, and obtained full root privileges on default installations of Ubuntu 20.04, Ubuntu 20.10, Ubuntu 21.04, Debian 11, and Fedora 34 Workstation . . .
To the best of our knowledge, this vulnerability was introduced in July 2014 . . .

`https://www.qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt`

Real world example 5: FORCEDENTRY (2021)

. . . a zero-day zero-click exploit against iMessage . . .
All iPhones with iOS versions prior to 14.8, All Mac computers with operating system versions prior to OSX Big Sur 11.6, Security Update 2021-005 Catalina, and all Apple Watches prior to watchOS 7.6.2.

. . .
The exploit works by exploiting an integer overflow vulnerability in Apple's image rendering library (CoreGraphics). We are publishing limited technical information about CVE-2021-30860 at this time.

https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/

Real world example 6: "BadAlloc" (2021, 1/2)

"BadAlloc" . . . wide range of IoT and OT devices in industrial, medical, and enterprise networks . . . memory allocation implementations written throughout the years . . . have not incorporated proper input validations. . . . resulting in execution of malicious code on a target device

https://msrc.microsoft.com/blog/2021/04/badalloc-memory-allocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/

```
void * calloc(size_t __nmemb,size_t __size)

{
  size_t __n;
  void *__s;

  __n = __size * __nmemb;
  if ((_func_memPartCacheAlloc == (code *)0x0) ||
     (__s = (void *)(*_func_memPartCacheAlloc)(memSysPartId,__n,memDefaultAlignment),
     __s == (void *)0x0)) {
    __s = (void *)memPartAlloc(memSysPartId,__n);
  }
  if (__s != (void *)0x0) {
    bzero(__s,__n);
  }
  return __s;
}
```

Real world example 6: “BadAlloc” (2021, 2/2)

```
void * pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t * pxBlock, * pxPreviousBlock, * pxNewBlockLink;
    static BaseType_t xHeapHasBeenInitialised = pdFALSE;
    void * pvReturn = NULL;

    vTaskSuspendAll();
    {
        /* If this is the first call to malloc then the heap will require
         * initialisation to setup the list of free blocks. */
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }

        /* The wanted size is increased so it can contain a BlockLink_t
         * structure in addition to the requested amount of bytes. */
        if( xWantedSize > 0 )
        {
            xWantedSize += heapSTRUCT_SIZE;

            /* Ensure that blocks are always aligned to the required number of bytes. */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                /* Byte alignment required. */
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize & portBYTE_ALIGNMENT_MASK ) );
            }
        }
```

0xffffffff + 8 = 7

An example of C peculiarity

A different example. . . what's wrong (*if any*) with this?

```
int bad_strcmp(const char *s1, const char *s2)
{
        while (*s1 && *s2 && *s1==*s2) {
                ++s1;
                ++s2;
        }
        return *s1 - *s2;
}
```

`https://pubs.opengroup.org/onlinepubs/009695399/functions/strcmp.html`

specifies that: "...the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ..."; however, the C standard [ANS11] says that: "The three types char, signed char, and unsigned char [...] The implementation shall define char to have the same range, representation, and behavior as either signed char or unsigned char."

SQL injection example (1/2)

```
# get the input from the user
name = ...
surname = ...

# build the SQL insert command
# eg.: INSERT INTO Students(name, surname) VALUES('foo', 'bar');
sql_command = "INSERT INTO Students(name, surname) VALUES('" +
                    name + "', '" + surname + "');"

# ... and send the command to the DB server
execute_sql(sql_command)
```

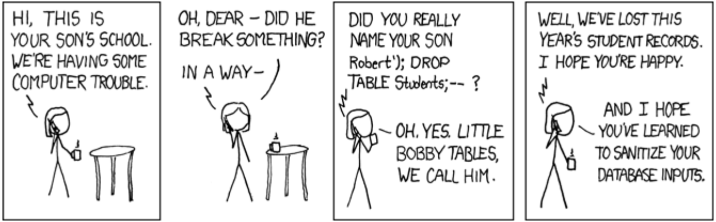
SQL injection example (2/2)

Consider:

```
name = "foo', 'bar'); DROP Students; -- "
surname = "sql injection example"
```

Then, the command becomes:

```
"INSERT INTO Students(name, surname) VALUES('foo', 'bar');
DROP Students; -- ', 'sql injection example');"
```



https://xkcd.com/327/

The underlying problem

mixing code and (user provided) data; to avoid these problems you can:

- sanitize/escape data
  - use standard functions, don't roll your own, and
  - hope they are correct (e.g. `htmlspecialchars` vulnerability `https://www.cvedetails.com/cve/CVE-2009-4142/`)
- use compiled queries
- use domain-specific libraries/frameworks; e.g. ORMs
  - EF `https://msdn.microsoft.com/en-us/data/ef.aspx`
  - SQL Alchemy `https://www.sqlalchemy.org/`
  - ...

You can also use sqlmap `https://sqlmap.org/` on your application to check for vulnerabilities

Command injection

```python
import os

while True:
    f = input('Enter the filter (none to list all processes): ')
    cmd = 'ps aux'
    if f:
        cmd += " | grep '{}'".format(f)
    os.system(cmd)
```

Log4Shell — CVE-2021-44228

- an injection vulnerability in Log4j, a popular logging framework
- the vulnerability had existed unnoticed since 2013 (!)
- simple exploit, estimated to affect hundreds of millions of devices
  - in the default configuration, when logging a string, Log4j performed string substitution on expressions of the form `${`*prefix*`:`*name*`}`
    - one recognized expression is `${jndi:`*lookup*`}`, where an arbitrary URL may be queried and loaded as Java object data; e.g., `${jndi:ldap://example.com/file}`
  - by inputting a string that is logged, an attacker could load and execute malicious code hosted on a public URL
  - log4j 2.15.0 disabled this behavior by default
  - 2.16.0 completely removed this functionality

`https://en.wikipedia.org/wiki/Log4Shell`

Stranger Strings — CVE-2022-35737

SQLite is used in nearly everything, from naval warships to smartphones to other programming languages. The open-source database engine has a long history of being very secure...

SQLite implements custom versions of the printf family of functions and adds the new format specifiers %Q, %q, and %w, which are designed to properly escape quote characters in the input string in order to make safe SQL queries ...

...is exploitable when large string inputs are passed to ...SQLite ...printf functions ...cause the program to crash ...if the format string contains the ! special character to enable unicode character scanning, then it is possible to achieve arbitrary code execution in the worst case, or to cause the program to hang ...

`https://blog.trailofbits.com/2022/10/25/sqlite-vulnerability-july-2022-library-api/`

Validation

- *all untrusted input* source must be validated/sanitized
- prefer allow-listing w.r.t deny-listing
  - deny-lists can be useful for (security) testing; e.g., Apple's "goto fail"

goto fail (2014 — Apple's SSL/TLS implementation)

```
int SSLVerifySignedServerKeyExchange(...)
{
        ...
        err = 0;
        ...
        if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
           goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
           goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
           goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
           goto fail;
           goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
           goto fail;
        ...
        fail:
        ...
        return err;
}
```

Memory corruption

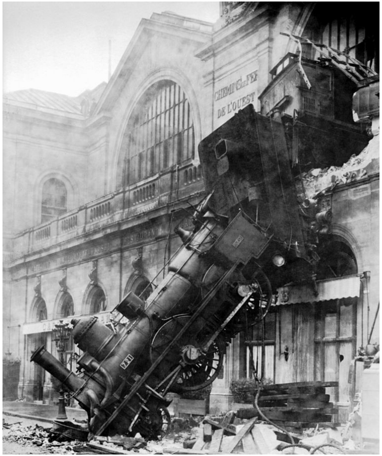
Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program's behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated. [SPWS13]

"more than 30" in 2013!

The most infamous bug is... the buffer overflow

What is a buffer overflow? (no technicalities)

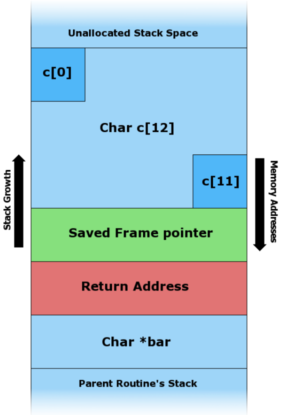
Exploiting a programming error, like this ☺:



October 22, 1895; `en.wikipedia.org/wiki/Montparnasse_derailment` (idea:[Whe16])

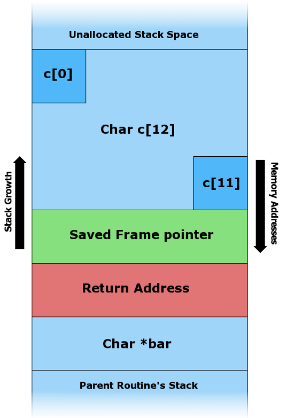
A classic one: stack-overflow (1/3)

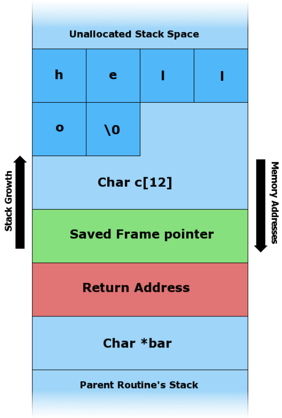
```c
#include <string.h>

void foo(char *bar) {
   char  c[12];
   strcpy(c, bar);
}

int main(int argc, char **argv) {
   foo(argv[1]);
}
```



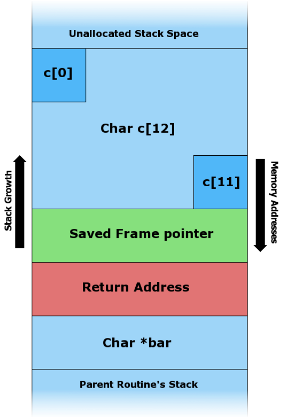
https://en.wikipedia.org/wiki/Stack_buffer_overflow

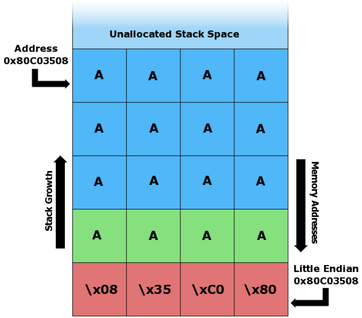
A classic one: stack-overflow (2/3)





A classic one: stack-overflow (3/3)





Taint Analysis

In which of the following examples should we worry and perform a taint analysis?

1. `read(fd, buf, 1024);`
2. `strcpy(dest, src);`
3. `read(fd, buf, len);`
4. `gets(buf);`
5. `fgets(buf, size, f);`

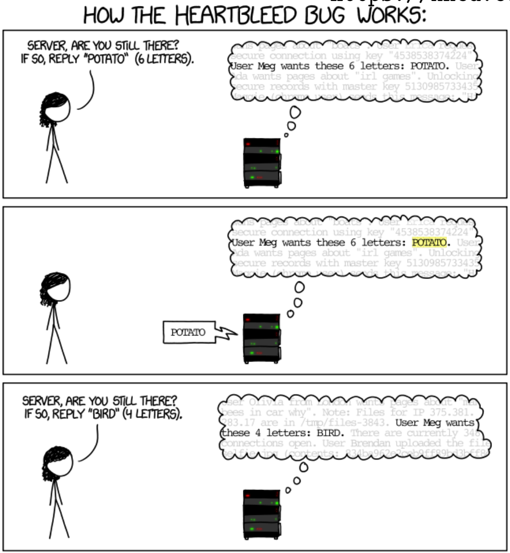
Read overflow

Rather than writing past the end of a buffer, a bug could permit *reading* past the end, leaking information

Hearthbleed (2014)

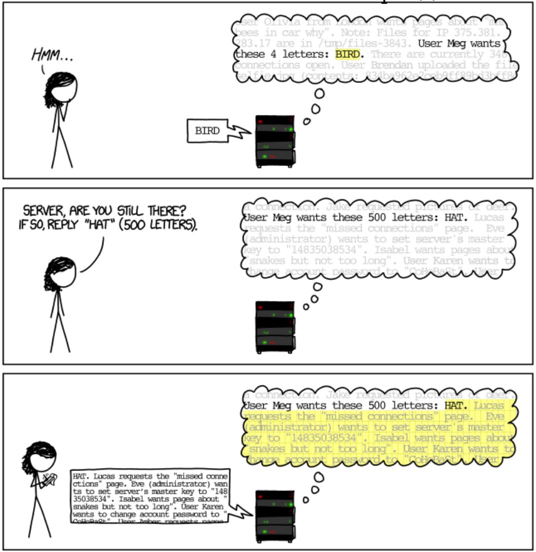
Heartbleed is a security bug disclosed in April 2014 in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol.
At the time of disclosure, some 17% (around half a million) of the Internet's secure web servers certified by trusted authorities were believed to be vulnerable to the attack, allowing theft of the servers' private keys and users' session cookies and passwords.
`https://heartbleed.com/`

https://xkcd.com/1354/



Heartbleed (2/2)

https://xkcd.com/1354/



What are the “inputs” of a program?

What can cause a change in a program behavior?

- Obvious ones
  - Command-line arguments
  - What the user types/clicks/...
  - File contents
- Not so obvious one: the environment the program runs in
  - the current directory
  - `PATH`
  - Linux: `LD_LIBRARY_PATH`/`LD_PRELOAD`
  - the inherited handles (file descriptors, sockets, ...)

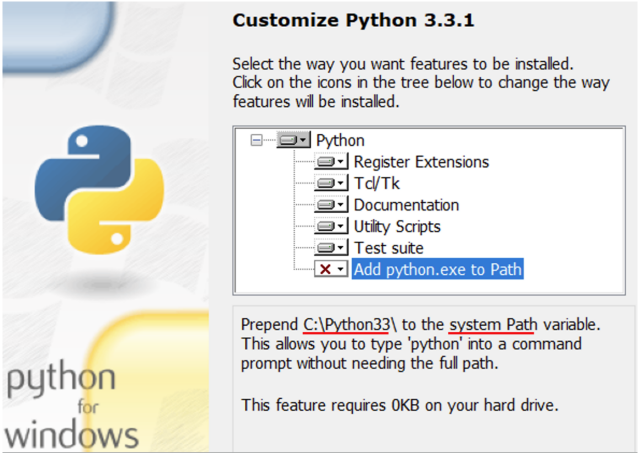
DLL Hijacking

> *By manipulating environment variables on process level, it is possible to let trusted applications load arbitrary DLLs and execute malicious code. This post lists nearly 100 executables vulnerable to this type of DLL Hijacking on Windows 11 (21H2)*
>
> `https://www.wietzebeukema.nl/blog/save-the-environment-variables`

See also: `https://hijacklibs.net/`

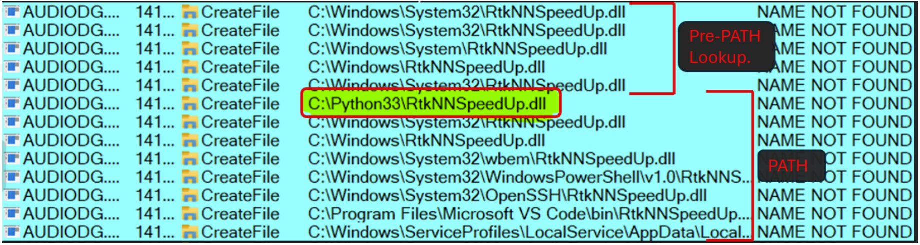
DLL search order include the current directory, and the `PATH` environment variable; if one of them is user-writable, an attacker can place a malicious DLL there and have it loaded by a legitimate application. . .

An example: Python v3.3.1 + Audio driver (1/2)



Python 3.3.1 adds a writable directory into the system path; the audio driver, a privileged process, looked for an unexisting DLL...

An example: Python v3.3.1 + Audio driver (2/2)



By providing a DLL named `RtkNNSpeedUp.dll`, an attacker could perform a privilege escalation attack

→`demo_audiodg.mp4`

Nebula

Nebula is a VM (an ISO file, actually) that takes you through a variety of weaknesses and vulnerabilities in Linux

`https://exploit.education/nebula/level-01/`
Level usernames *and* passwords: `level`*xx*

$\rightarrow$ `examples/nebula1.c`

TOCTTOU (1/2)

Consider the following snippet:

- assume it is contained in a setuid (root) program
- note the use of `access`, to check whether the real user would be allowed to write the file

```
if (access("file", W_OK) != 0)
   exit(EXIT_FAILURE);
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- do you see any problem?
- does `access` really use the *real* user for checking?

TOCTTOU (2/2)

Time of check to time of use is a class of bugs caused by changes in a system between the checking of a condition (such as a security credential) and the use of the results of such a check

```
/* victim */
if (access("file", W_OK) != 0)
   exit(EXIT_FAILURE);



fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

```
/* attacker */


symlink("/etc/passwd",
              "file");
```

Since Linux 3.6, symlink protection enabled by default

See `/proc/sys/fs/protected_symlinks` in `proc(5)`

Dirty Cow

Dirty COW

A race condition was found in the way the Linux kernel's memory subsystem handled the copy-on-write (COW) breakage of private read-only memory mappings. An unprivileged local user could use this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system.

DIRTY COW

`https://dirtycow.ninja`

Existed in the Linux kernel since version 2.6.22 released 2007, and there is information about it being actively exploited at least since October 2016

Example: Compilation service

Consider the following situation:

- A program provides compilation services to other programs
- The client program specifies the name of the input and output files
- the server is given the same access to those files that the client has
- however, the compiler service is pay-per-use, and the compiler service stores its billing information in a file (dubbed BILL) that only it has access to

Does it seem reasonable?

- Suppose a client calls the service and names its output file BILL
- The service opens the output file. Even though the client did not have access to that file, the service does, so the open succeeds, and the server writes the compilation output to the file, overwriting it, and thus destroying the billing information [Har88]

`https://en.wikipedia.org/wiki/Confused_deputy_problem`

Principle of the least privilege

Principle of the least privilege

Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

Jerome Saltzer, Communications of the ACM

The oldest instance of least privilege is probably the source code of `login.c`

- begins execution with super-user permissions and
- the instant they are no longer necessary, dismisses them via `setuid` with a non-zero argument
- as demonstrated in the Version 6 Unix source code:
  `https://www.retro11.de/ouxr/u6ed/usr/source/s1/login.c.html#n:132`

From Wikipedia: `https://en.wikipedia.org/wiki/Principle_of_least_privilege`

Principle of the least privilege in OSes

Can you think of applications of this principle in modern OSes?

Linux sudo `https://en.wikipedia.org/wiki/Sudo`

Unfortunately, dropping unneeded privileges is error-prone, and there are portability issues [TDSW08]; moreover, we may hope sudo to be bug-free, but various bugs have been found (and fixed).

See also, *Zero-day vulnerability in Bash - Suidbash Google CTF Finals 2019* `https://www.youtube.com/watch?v=-wGtxJ8opa8`

Windows UAC (User Account Control)

`https://en.wikipedia.org/wiki/User_Account_Control`

unfortunately, useless with default settings (!) see `https://github.com/hfiref0x/UACME` and `devblogs.microsoft.com/oldnewthing/20160816-00/?p=94105`

Format string vulnerabilities

Just the idea; see [stt01, HLV10] for more

```
#include <stdio.h>

int main(int argc, char* argv[])
{
        int i;
        for(i = 1; i < argc; ++i)
                printf(argv[i]);
        printf("\n");
}
```

Think of %x, %p, or %s; moreover. . . and %n allows us to *write* to memory

Bad strcmp revisited

```
int still_bad_strcmp(const char *s1, const char *s2)
{
        while (*s1 && *s2 && *s1==*s2) {
                ++s1;
                ++s2;
        }
        return *((unsigned char *)s1) - *((unsigned char *)s2);
                // (unsigned) chars are promoted
                // to int to perform the subtraction
}

bool check_password(const char *password)
{
        return still_bad_strcmp(password, "zxgio") == 0;
}
```

...what's wrong here?

Side channels (1/2)

Side channel attacks

In computer security, a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs). Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.

`https://en.wikipedia.org/wiki/Side-channel_attack`

Side channels (2/2)

Also:

Side channel = "Obtaining meta-data and deriving data from it."

by Daniel Gruss (@lavados)

Thread: `https://twitter.com/lavados/status/1156982866414379008`

Important recent examples are Meltdown [LSG+18], Spectre [KHF+19] and Bleichenbacher's CAT[RGG+18]

Outline

Introduction

How can we find/avoid vulnerabilities?

There are many techniques and tools, with no clear winners between:

- Static Analysis
  - Can work on incomplete code
  - A single run can analyze all code
  - However, there are false positives and false negatives
- Dynamic Analysis
  - Usually, easy to set up and run
  - No false positives
  - No false negatives *but* limited to what has been executed
  - Slows down execution

Static analysis tools

- behave a bit like spell-checkers
  - they prevent well-understood varieties of mistakes from going unnoticed
  - a clean run doesn't guarantee that code is perfect
    - it is probably just free of certain kinds of common problems
  - however, they don't automatically make you an excellent coder ☺
- can find errors early in development, even before the first run
- can recheck large bodies of code when a new attack is discovered

No silver bullets

The problem with *interesting* static analyses is that they are undecidable; in practice, most "work" (=produce useful results) but:

- false positives/alarms
- false negatives — unreported problems that exist in the program

The balance between them is often indicative of the purpose of the tool:

- code quality tools usually produce a low number of false positives
- security tools usually produce more false positives

Static analysis tools check the code (only)

To catch a defect, it must be "visible" in the code: architectural risk analysis is a necessary complement

Type checking

In Java, for instance, we have both false positives:

```java
short s = 0;
int i = s;
short r = i; // error: incompatible types: possible lossy
             // conversion from int to short
```

And false negatives:

```java
Object[] objs = new String[1];
objs[0] = new Object(); // Exception in thread "main"
                        // java.lang.ArrayStoreException:
                        // java.lang.Object
```

Program verification

Program verification tools try to prove that some code is a faithful implementation of a specification

- creating proper specifications can require more work than writing code
- historically, these tools could not process programs of significant size

More commonly, verification tools check software against a *partial* specification that details only part of the behavior of a program. This is sometimes called property checking
For instance, . . .

Property checking example

For instance, properly allocating/releasing memory:

```
int f()
{
char *inBuf, *outBuf;
inBuf = malloc(512);
outBuf = malloc(512);
if (outBuf == 0 || inBuf == 0)
        return -1; // [...] A memory leak is possible.
```

Ok; and now?

```
int f()
{
  char *inBuf, *outBuf;
  inBuf = malloc(512);
  if (inBuf == 0)
    return -1;  // as before... (for some tools)
```

Bug finding

A middle ground between style-checking and program verification

A bug finder points out places where the program will behave in a way that the programmer did not *probably* intend

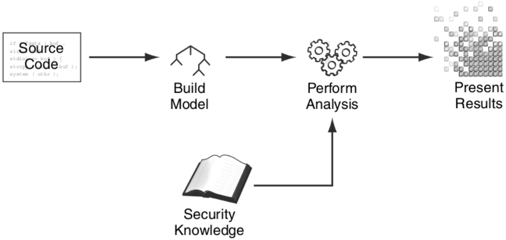
- most come pre-stocked with a set of "bug idioms" (rules) that describe patterns that often indicate bugs

E.g. CLang Static Analyzer or FindBugs (Java)

Inner working

How do these tool work? The first thing is transforming the code to be analyzed into a model, that is, a set of data structures

- the kind of model depends on the analysis
- many data-structures and algorithms shared with compiler world



From [CW07]

Reporting results

If you can't make sense of what a tool reports, the result is useless; e.g.

- too many false positives
- bad presentation of "good" results can be confused with analysis mistakes

tools should offer

- integration with IDEs
- a way to *easily*
  - navigate, group and sort results (by severity, confidence, ...)
  - eliminate/suppress unwanted warnings
  - explain the reason for each warning
  - provide recommendations

Crucial issue: context sensitivity

Context sensitivity, that is, circumstances and conditions under which a particular piece of code runs is crucial:

- Easy to point at all calls to `strcpy`
- Hard to pinpoint specific calls that might allow an attacker to overflow a buffer

E.g.

```
int main(int argc, char **argv)
{
        char buf1[1024];
        char buf2[1024];
        strcpy(buf1, "pizza");
        strcpy(buf2, argv[0]);
...
```

How to use static-analysis tools

Some tools are simply run like compilers

- i.e , you run them on source files and get the list of warnings in return

Others, require a multi-step approach

- a command “monitors” the building process, producing either
  - the analysis result, or
  - some intermediate files; that are then processed by another command, the *analyzer*
- another command shows the results or export them in a (PDF/HTML/...) report

CppCheck

Cppcheck is a static analysis tool for C/C++ code. It focuses on detecting undefined behaviour and dangerous coding constructs. The goal is to detect only real errors in the code (i.e. have very few false positives).

`https://cppcheck.sourceforge.net/`

- Installation: `sudo apt install cppcheck`
- Usage: `cppcheck` *`file-or-path`*

```
kill.c:117:11: style: Local variable 'rv' shadows outer variable [shadowVariable]
      int rv = print_signal_list();
          ^
kill.c:106:7: note: Shadowed declaration
  int rv = 0;
      ^
kill.c:117:11: note: Shadow variable
      int rv = print_signal_list();
          ^
shell.c:106:27: style: The scope of the variable 'prev' can be reduced. [variableScope]
  procedure *cur = proc, *prev;
                          ^
tokenizer.c:96:17: style: The scope of the variable 'repl' can be reduced. [variableScope]
          char* repl;
                ^
```

Clang Static Analyzer

Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs

`https://clang-analyzer.llvm.org`

- Installation: `sudo apt install clang-tools`
- Usage:
  - `scan-build` monitors the build process
  - `scan-view` creates a web-server that allows you to browse the issues

For instance, ...

Clang Static Analyzer example (1/2)

Bug Summary

Bug Type	Quantity	Display?
All Bugs	**17**	☑
Dead store		
Dead assignment	1	☑
Logic error		
Dereference of undefined pointer value	2	☑
Memory error		
Double free	2	☑
Memory leak	5	☑
Security		
Potential insecure memory buffer bounds restriction in call 'strcat'	2	☑
Potential insecure memory buffer bounds restriction in call 'strcpy'	5	☑

Reports

Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length			
Dead store	Dead assignment	tokenizer.c	tokenize	195	1	View Report	Report Bug	Open File
Logic error	Dereference of undefined pointer value	shell.c	build_procedure_list	90	20	View Report	Report Bug	Open File
Logic error	Dereference of undefined pointer value	shell.c	build_procedure_list	93	17	View Report	Report Bug	Open File
Memory error	Double free	tokenizer.c	tokenize	182	38	View Report	Report Bug	Open File
Memory error	Double free	tokenizer.c	tokenize	115	25	View Report	Report Bug	Open File

Clang Static Analyzer example (2/2)

```
else{
  char* envVar = strndup(&line[i+1], j-i-1);
  char* variable_value = getenv(envVar);

  if(variable_value)
```

31 ← Assuming 'variable_value' is non-null →

32 ← Taking true branch →

```
    repl = variable_value;
  else
    repl = getsudoenv(envi, envVar);

  strcpy(&token[n], repl);
  n += strlen(repl);
  free(repl);
```

33 ← Memory is released →

```
  free(envVar);
  free(variable_value);
```

34 ← Attempt to free released memory

```
}
```

PVS-Studio

PVS-Studio is a tool for detecting bugs and security weaknesses in the source code of programs, written in C, C++, C# and Java.

- Commercial, but free for students: `https://www.viva64.com/en/for-students/`
- Usage:
  - `pvs-studio-analyzer trace` monitors the build process
  - `pvs-studio-analyzer analyze` analyzes ☺
  - `plog-converter` exports in various formats

For instance, ...

PVS-Studio example (1/2)

PVS-Studio Analysis Results

Date:	Thu Nov 19 12:25:33 2020
PVS-Studio Version:	7.10.43305.85
Command Line:	plog-converter -t fullhtml -a GA\;64\;OP\;CS shell.log -o pvs-shell-analysis
Total Warnings (GA):	27

Group	Location	Level	Code	
General Analysis	execute.c:62	Medium	V522	Dereferencing of the null pointer 'path' might take place. The potential null pointer is passed into 'check_file' function. Inspect
General Analysis	execute.c:363	Medium	V575	The potential null pointer is passed into 'strchr' function. Inspect the first argument. Check lines: 363, 362.
General Analysis	export.c:11	Medium	V575	The potential null pointer is passed into 'strchr' function. Inspect the first argument. Check lines: 11, 10.
General Analysis	pwd.c:9	Medium	V575	The potential null pointer is passed into 'getcwd' function. Inspect the first argument. Check lines: 9, 8.
General Analysis	pwd.c:12	High	V575	The null pointer is passed into 'free' function. Inspect the first argument.
General Analysis	shell.c:87	Medium	V522	There might be dereferencing of a potential null pointer 'cur'. Check lines: 87, 78.
General Analysis	shell.c:128	Medium	V755	A copy from unsafe data source to a buffer of fixed size. Buffer overflow is possible.
General Analysis	shell.c:136	Medium	V755	A copy from unsafe data source to a buffer of fixed size. Buffer overflow is possible.
General Analysis	shell.c:148	Medium	V728	An excessive check can be simplified. The '||' operator is surrounded by opposite expressions '!loop' and 'loop'.
General Analysis	sudo_environment.c:19	Medium	V701	realloc() possible leak: when realloc() fails in allocating memory, original pointer 'e->elems' is lost. Consider assigning realloc(
General Analysis	sudo_environment.c:23	Low	V522	There might be dereferencing of a potential null pointer 'e->elems'.

PVS-Studio example (2/2)

```
int main(unused int argc, unused char *argv[]) {
  init_shell();

  static char line[4096];
  int line_num = 0;

  int loop = true;

  if(argc > 1){
    if(strcmp(argv[1], "-c") == 0){
      loop = false;
      strcpy(line, argv[2]);
```

↑ V755 A copy from unsafe data source to a buffer of fixed size. Buffer overflow is possible.

```
    }else{
      fprintf(stderr, "wrong flag");
    }
  }
```

Facebook Infer

Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs.

`https://fbinfer.com/`
`https://github.com/facebook/infer/releases`

For instance, with this program:

```c
#include <stdlib.h>

void test() {
  int *s = NULL;
  *s = 42;
}
```

you get:

```
hello.c:5: error: NULL_DEREFERENCE
  pointer s last assigned on line 4 could be null and is dereferenced at line 5
```

Outline

Query languages

- Joern, which leverages *Code Property Graphs* [YGAR14]
  `https://joern.io/`
- SemGrep
  `https://semgrep.dev/`
- CodeQL
  `https://securitylab.github.com/tools/codeql`
- ...

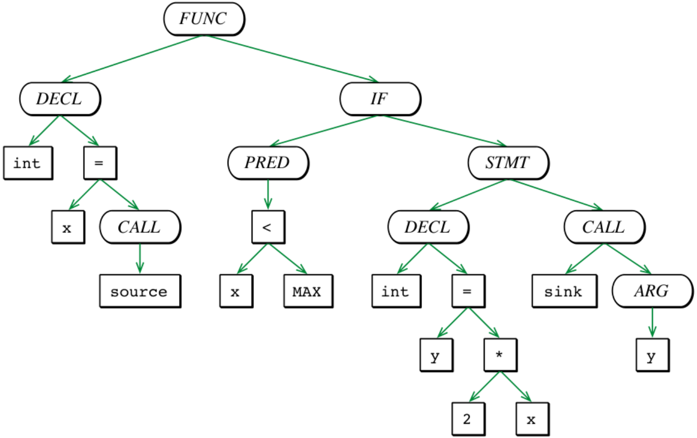
Code Property Graphs: code sample

Example taken from the original paper [YGAR14]

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

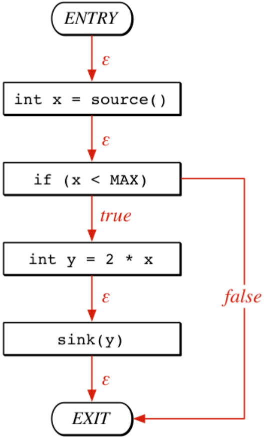
Code Property Graphs: Abstract Syntax Trees

ASTs are ordered trees: inner nodes represent operators and leaf nodes correspond to operands
Neither control flow nor data dependencies are encoded in this representation



Code Property Graphs: Control Flow Graphs

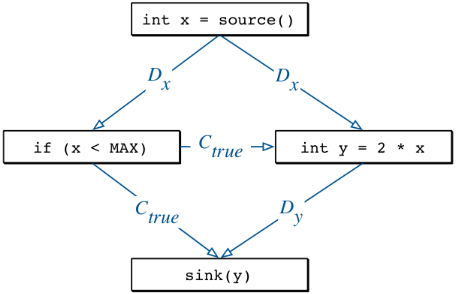
CFGs model the order in which code statements are executed, and conditions that need to be met for a particular path of execution to be taken



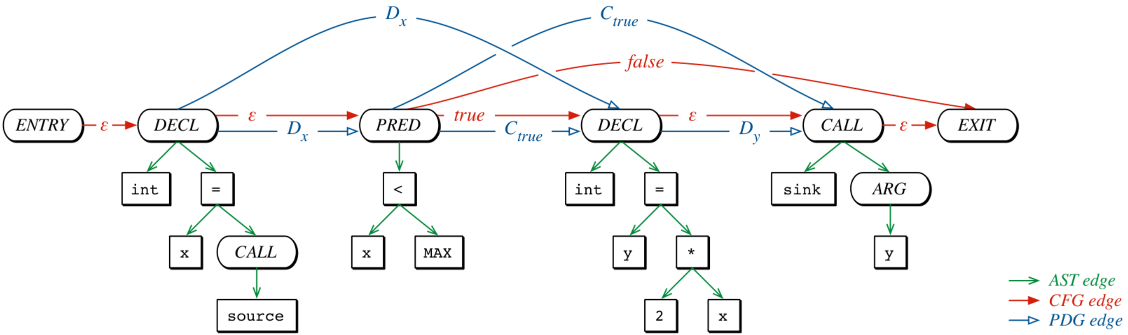
Code Property Graphs: Program Dependence Graphs

PDGs use wo types of edges: data dependency, reflecting the influence of one variable on another, and control dependency corresponding to the influence of predicates on the values of variables

Note: PDGs are derived from CFGs, but the execution order is lost



Code Property Graphs: putting everything together



- CPGs enable modeling patterns for common vulnerabilities in terms of graph traversals
- [YGAR14] reports 18 previously unknown vulnerabilities in the Linux kernel
- queries can be tailored to identify vulnerabilities specific to a code base; examples can be found at `https://queries.joern.io/`

Dynamic code analysis

Dynamic analysis tools work by

1. (statically or dynamically) instrumenting the target application
2. observing an execution to detect errors

Major players: Valgrind, Dr.Memory and clang/gcc sanitizers

Valgrind

Instrumentation framework for building dynamic analysis tools, can detect many memory management and threading bugs, and profile your programs

- Works on Unix-like OSes (i.e. no Windows)
- Runs unmodified binaries; however, it's better to
  - compile with symbols (-g) and
  - without/with-little optimizations; otherwise, valgrind occasionally reports spurious errors
- Only for heap allocations (i.e. no stack/global)
- Huge slow-down (20-30x)
- Installation: `apt install valgrind`
- Quick-start: compile with debug symbols, then

  ```
  valgrind [--leak-check=full] executable args # memory
  valgrind --tool=helgrind executable args # threading
  ```
- Offers a gdb-server:

  ```
  valgrind --vgdb=full --vgdb-error=0 executable args
  ```

https://valgrind.org

Dr. Memory

Similarly to Valgrind, identifies memory-related programming errors

- Works on Windows, Linux (+Android), Mac
- Runs unmodified binaries (but same considerations as valgrind)
- Generally faster than Valgrind
- Distributed as a `tar.gz`
- Quick-start:
  `g++ -m32 -g -fno-inline -fno-omit-frame-pointer ... drmemory -- `*`executable args`*

`https://drmemory.org`

Outline

Sanitizers

We'll consider LLVM Sanitizers [SBPV12]; for a full picture see also [SLR+18]

- use compile-time instrumentation (Clang, some work in gcc too)
- detect memory, thread errors and undefined behaviors
- way faster than Valgrind/Dr.Memory

they are:

- AddressSanitizer (+LeakSanitizer) — addressability issues and memory leaks
  - On some platform there is also a more efficient Hardware-assisted AddressSanitizer
- MemorySanitizer — use of uninitialized memory
- ThreadSanitizer — data races and deadlocks
- UndefinedBehaviorSanitizer – (some) undefined behaviors

Recently introduced into Visual Studio too:
`https://learn.microsoft.com/en-us/cpp/sanitizers/asan`

Address Sanitizer

Finds

- buffer overflows (stack, heap, globals)
- heap-use-after-free, stack-use-after-return
- leaks, init-order, double-free, ...

by

- instrumenting all loads/stores, and replacing `malloc` and friends

To use:

```
-fsanitize=address -g -fno-common -fno-omit-frame-pointer
```

Beware:

1. `-fsanitize=address` must be used both when compiling and linking; i.e. `C(PP)FLAGS` and `LDFLAGS`
2. uses `ptrace`: use `ASAN_OPTIONS=detect_leaks=0 gdb ...` to debug (don't `export`, otherwise detection would be always disabled)

Special checks can be enabled via `ASAN_OPTIONS`; see `https://github.com/google/sanitizers/wiki/AddressSanitizerFlags`

How does this work?

The instrumentation replaces (de)allocation functions, that

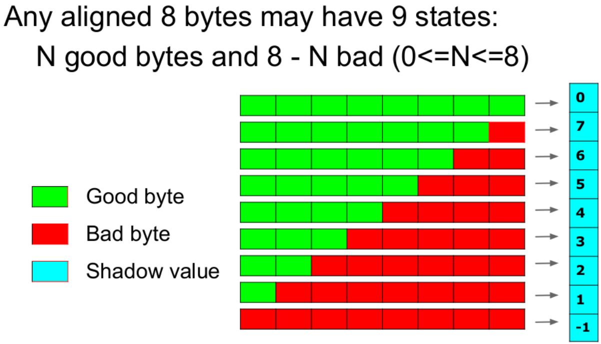
- insert redzones around every allocation, as we'll discuss
- delay the reuse of freed memory
  - poisoning the entire memory region on `free`
- collect stack traces for every `malloc`/`free`

Overhead

- 2x slowdown
- 1.5x-3x memory

Shadow map

Part of the address space is reserved for a shadow map, that encodes the state of other parts



Instrumentation

```
uint64_t *a = ...
*a = ... // 8-byte access
```

$\Downarrow$

```
char *shadow = (a >> 3) + shadowOffset;
if (*shadow)
        ReportError(a);
*a = ...
```

accessing $1, 2, 4$ bytes is a bit more involved (same idea, though)

Instrumenting stack frames

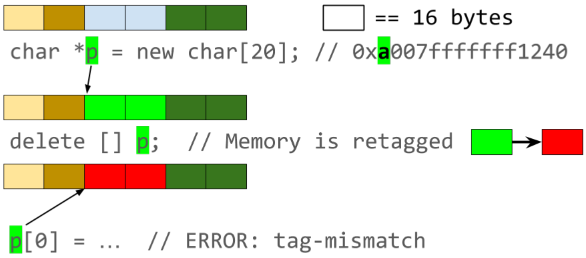
```
void foo() {
        char a[328];
        <------------- CODE -------------> }
```

$$\Downarrow$$

```
void foo() {
        char rz1[32]; // 32-byte aligned
        char a[328];
        char rz2[24];
        char rz3[32];
        int *shadow = (&rz1 >> 3) + shadowOffset;
        shadow[ 0] = 0xffffffff; // poison rz1
        shadow[11] = 0xffffff00; // poison rz2
        shadow[12] = 0xffffffff; // poison rz3
        <------------- CODE ------------->
        shadow[0] = shadow[11] = shadow[12] = 0;
}
```

HW-assisted Address Sanitizer (1/2)

HWASan is based on memory tagging . . . Every memory allocation is assigned a random 8-bit tag that is stored in the most significant byte (MSB) of the address, but ignored by the CPU



Better granularity and performance (especially when we'll have fully hardware-supported memory tagging, as in ARM v8.5)

HW-assisted Address Sanitizer (2/2)

Resources:

- Memory Tagging for the Kernel: Tag-Based KASAN by Andrey Konovalov @ Android Security Symposium 2020
  https://www.youtube.com/watch?v=f-Rm7JFsJGI
- Hardware-Assisted Address Sanitizer (HWASan) in Android
  https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html
- Memory Tagging and how it improves C/C++ memory safety by Kostya Serebryany @ CppCon 2018
  https://www.youtube.com/watch?v=lLEcbXidK2o

Memory Sanitizer

Detects uninitialized memory reads

Idea:

- Bit to bit shadow mapping (1 means *poisoned*; that is, uninitialized)
  - e.g., if `foo` is uninitialized, then `{foo &= 1;}` zero-initializes all its bits except for the least significant one!
- Memory returned by `malloc` and stack objects are *uninitialized*
- Shadow is unpoisoned when constants are stored
- MSan requires to recompile all libraries (to avoid false positives)
  - Libc can be wrapped, but inline-assembly and JIT?

`-fsanitize=memory`

`-fsanitize=memory -fsanitize-memory-track-origins`

Overhead

- Without origins: 2.5x slowdown, 2x memory
- With origins: 5x slowdown, 3x memory

Thread Sanitizer

Detects data races and deadlocks

- Compile-time instrumentation (LLVM, GCC)
  - Intercepts all reads/writes
- Run-time library
  - Replaces/intercept memory/synchronization functions

`-fsanitize=thread`

Overhead/limitations

- Only 64-bit Linux, does not instrument libraries and inline assembly
- 4x-10x slowdown (still way faster than helgrind), 5x-8x memory overhead

Undefined-behaviour Sanitizer

Modifies the program at compile-time, to catch various kinds of undefined behavior:

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination
- ...

See the list of available checks at
`https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

`-fsanitize=undefined`

Overhead

- $0 - 0.5x$ slowdown

Outline

Introduction

It was a dark and stormy night. Really. Sitting in my apartment in Madison in the Fall of 1988, there was a wild midwest thunderstorm pouring rain and lighting up the late night sky. That night, I was logged on to the Unix systems in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running. It was a race to type an input line before the noise overwhelmed the command. This fighting with the noisy phone line was not surprising. What did surprise me was the fact that the noise seemed to be causing programs to crash. And more surprising to me was the programs that were crashing—common Unix utilities that we all use everyday.

Barton Miller

from the book "Fuzzing for Software Security Testing and Quality Assurance"

History

The term fuzzing originates from those days, and those ideas generated a stream of research, e.g. [MFS90]; however:

- Testing programs with random inputs dates back to the 1950s, when data was still stored on punched cards
- The execution of random inputs is also called random testing or monkey testing

Generating Software Tests – Breaking Software for Fun and Profit

Very interesting resource, a “Textbook for Paper, Screen, and Keyboard” can be found at: `https://www.fuzzingbook.org/`

Fuzz-testing, AKA Fuzzing

- Basic idea: throwing "random garbage" into programs and make them crash ... in interesting and different ways
  - sanitizers can be very useful
- How to generate "garbage"? More importantly, "quality garbage"?
  - Throw a coin, repeatedly ☺
    - Randomness is ok, totally random inputs may be useless
    - Inputs should be "representative" of expected format
  - Generate random test-case from a model; e.g. from grammars [GZ19]
  - Randomly mutate legal inputs; e.g. Radamsa `https://gitlab.com/akihe/radamsa`
  - Leverage code-coverage: e.g. using genetic algorithms to automatically discover clean, interesting test cases that trigger *new* internal states in the binary → AFL

American Fuzzy Lop (fuzzer)

AFL by Michal Zalewski

- is a security-oriented fuzzer
- employs compile-time instrumentation
  - on Linux, optional QEMU mode allows black-box binaries to be fuzzed
- uses genetic algorithms to discover interesting test cases, that trigger new internal states in the targeted binary

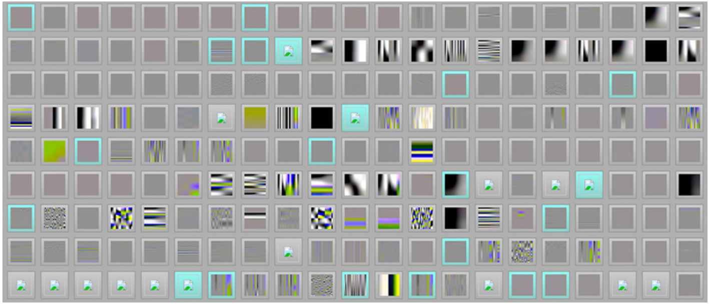
(Simplified) overall algorithm

1. load user-supplied initial test cases into the queue
2. take next input file from the queue
3. attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program
4. repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies
5. if any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue
6. go to (2)

The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds

Pulling JPEGs out of thin air

...created a text file containing just "hello" and asked the fuzzer to keep feeding it to a program that expects a JPEG image ... The first image, hit after about six hours on an 8-core system, looks very unassuming ... But the moment it is discovered, the fuzzer starts using the image as a seed - rapidly producing a wide array of more interesting pics ...



`https://lcamtuf.blogspot.it/2014/11/pulling-jpegs-out-of-thin-air.html`

How to get AFL

- Official repository: `https://github.com/Google/afl`
- AFL++ [FMEH20] is a superior fork to Google's AFL: more speed, more and better mutations, more and better instrumentation, custom module support, etc.
  `https://github.com/AFLplusplus/AFLplusplus`

References I

[ANS11] ANSI/ISO.
Working draft, Standard for Programming Language C.
Technical Report N1570, ANSI/ISO, 2011.

[ANS12] ANSI/ISO.
Working draft, Standard for Programming Language C++.
Technical Report N3337, ANSI/ISO, 2012.

[CW07] Brian Chess and Jacob West.
Secure programming with static analysis.
Pearson Education, 2007.

[DMS06] Mark Dowd, John McDonald, and Justin Schuh.
The art of software security assessment: Identifying and preventing software vulnerabilities.
Pearson Education, 2006.

References II

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse.
AFL++: Combining incremental steps of fuzzing research.
In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[GZ19] Rahul Gopinath and Andreas Zeller.
Building fast fuzzers, 2019.

[Har88] Norm Hardy.
The confused deputy (or why capabilities might have been invented).
ACM SIGOPS Operating Systems Review, 22(4):36–38, 1988.

[HLV10] Michael Howard, David LeBlanc, and John Viega.
24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.
McGraw-Hill, Inc., 2010.

References III

[KHF⁺19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.
Spectre attacks: Exploiting speculative execution.
In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[LH02] David LeBlanc and Michael Howard.
Writing secure code.
Pearson Education, 2002.

[LSG⁺18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg.
Meltdown: Reading kernel memory from user space.
In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

References IV

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So.
An empirical study of the reliability of unix utilities.
Communications of the ACM, 33(12):32–44, 1990.

[One96] Aleph One.
Smashing the stack for fun and profit.
http://insecure.org/stf/smashstack.html, 1996.

[Pol17] Erik Poll.
Lecture notes on language-based security, 2017.

[RGG+18] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom.
The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations, 2018.
http://cat.eyalro.net/.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov.
AddressSanitizer: A Fast Address Sanity Checker.
In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[SLR+18] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz.
SoK: Sanitizing for Security.
arXiv preprint arXiv:1806.04355, 2018.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song.
Sok: Eternal war in memory.
In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

References VI

[stt01] scut / team teso.
Exploiting format string vulnerabilities.
https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf, 2001.

[TDSW08] Dan Tsafrir, Dilma Da Silva, and David Wagner.
The murky issue of changing process identity: revising "setuid demystified".
USENIX Login, 33(3):55–66, 2008.

[WCC+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek.
Undefined behavior: what happened to my code?
In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[Whe16] David A. Wheeler.
Secure programming HOWTO, 2016.
http://www.dwheeler.com/secure-programs/.

References VII

[YGAR14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck.
Modeling and discovering vulnerabilities with code property graphs.
In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.