October 15, 2024

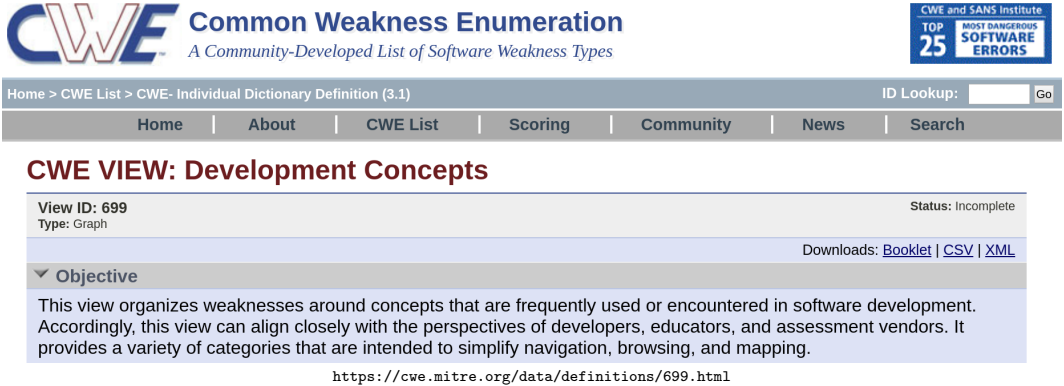
Software Security I&II

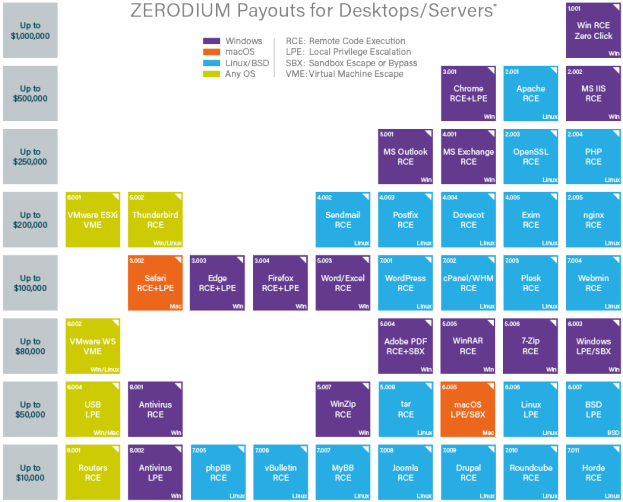
Rasmus Dahlberg

Learning outcomes

Already covered here or elsewhere: least privilege, modern crypto, use secure APIs, pass strings to complex subsystems with care, do unit testing, security audits, ...

Be aware of other common coding mistakes not covered here





https://zerodium.com/program.html

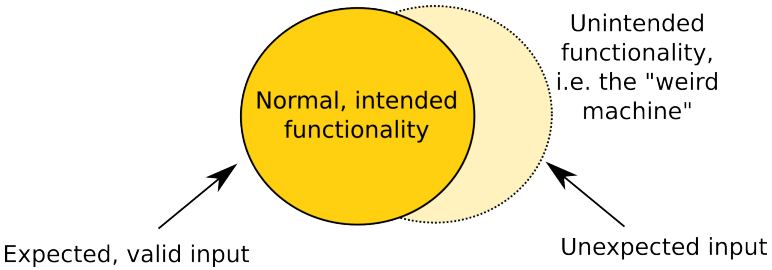
Reward amounts for security vulnerabilities

New! To read more about our approach to vulnerability rewards you can read our Bug Hunter University article here.

Rewards for qualifying bugs range from $100 to $31,337. The following table outlines the usual rewards chosen for the most common classes of bugs:

Category	Examples	Applications that permit taking over a Google account [1]	Other highly sensitive applications [2]	Normal Google applications	Non-integrated acquisitions and other sandboxed or lower priority applications [3]
Vulnerabilities giving direct access to Google servers					
Remote code execution	*Command injection, deserialization bugs, sandbox escapes*	$31,337	$31,337	$31,337	$1,337 - $5,000
Unrestricted file system or database access	*Unsandboxed XXE, SQL injection*	$13,337	$13,337	$13,337	$1,337 - $5,000
Logic flaw bugs leaking or bypassing significant security controls	*Direct object reference, remote user impersonation*	$13,337	$7,500	$5,000	$500
Vulnerabilities giving access to client or authenticated session of the logged-in victim					
Execute code on the client	Web: *Cross-site scripting* Mobile / Hardware: *Code execution*	$7,500	$5,000	$3,133.7	$100
Other valid security vulnerabilities	Web: *CSRF, Clickjacking* Mobile / Hardware: *Information leak, privilege escalation*	$500 - $7,500	$500 - $5,000	$500 - $3,133.7	$100

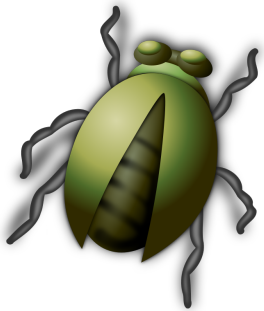
https://www.google.com/about/appsecurity/reward-program/index.html



Security properties and threat model → program should work as intended

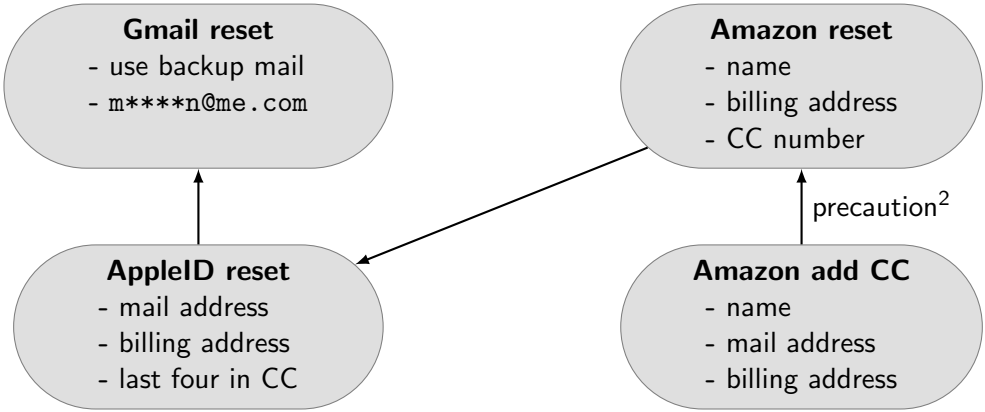
[1] https://en.wikipedia.org/wiki/Weird_machine

Can you spell software vulnerability? BUG



Bad composition

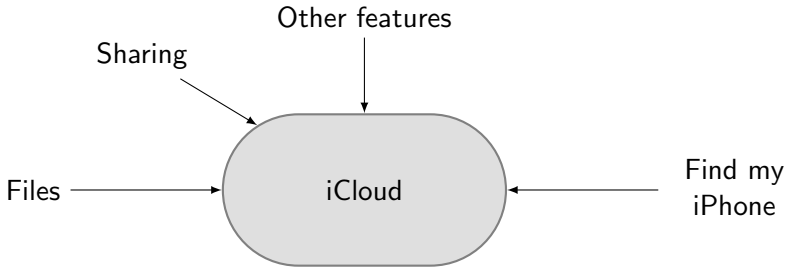
Gmail password reset[3]



[2] Obfuscate CC, s.t., only last last 4 CC digits are shown

[3] https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/

Credit: Nickolai Zeldovich and James Mickens



- User must login to use a feature
- **Rate limited login attempts?**

Files, sharing, other features? Yep
Find my iPhone...? Nop

Lesson learned: the importance of testing against abnormal behaviour

[4] https://github.com/hackappcom/ibrute

Credit: Nickolai Zeldovich and James Mickens

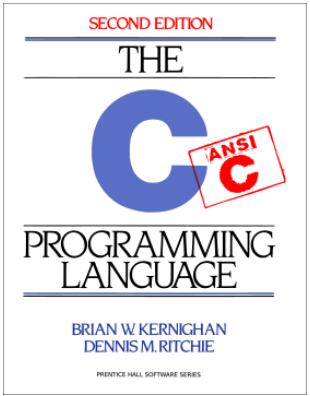
- TLS certificate: identity-to-key binding
- Subject name? Pascal string
  - Length followed by characters
- Many TLS implementations? C string
  - Characters with null-termination

Lesson learned: only process data at uniform formats

[5] https://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-PAPER1.pdf

The bad news :/

- Much software is written in C/C++
- Recipe for disaster:
  - Exposure to raw memory addresses
  - No built-in bound checking and safety
  - Operate on untrusted user input
- Why?



It is easy to get wrong


```c
char b[4] = "abc";
b[3] = 'd';
printf("b: %s\n", b);
...
```

Problem?
over-read

```c
char b[4] = "abc";
b[4] = 'd';
printf("b: %s\n", s);
...
```

Problem?
over-write

```c
char b[4];
fgets(b, 4, stdin);
printf(b);
...
```

Problem?
over-read

It is easy to get wrong


```
1  void init(char v, char *buf, int n)
2  {
3    char *b = buf;
4    while (b < buf+n) {
5      *b++ = val;
6    }
7  }
8  ...
9  char *buf = malloc(2);
10 init('A', buf, sizeof(buf));
```

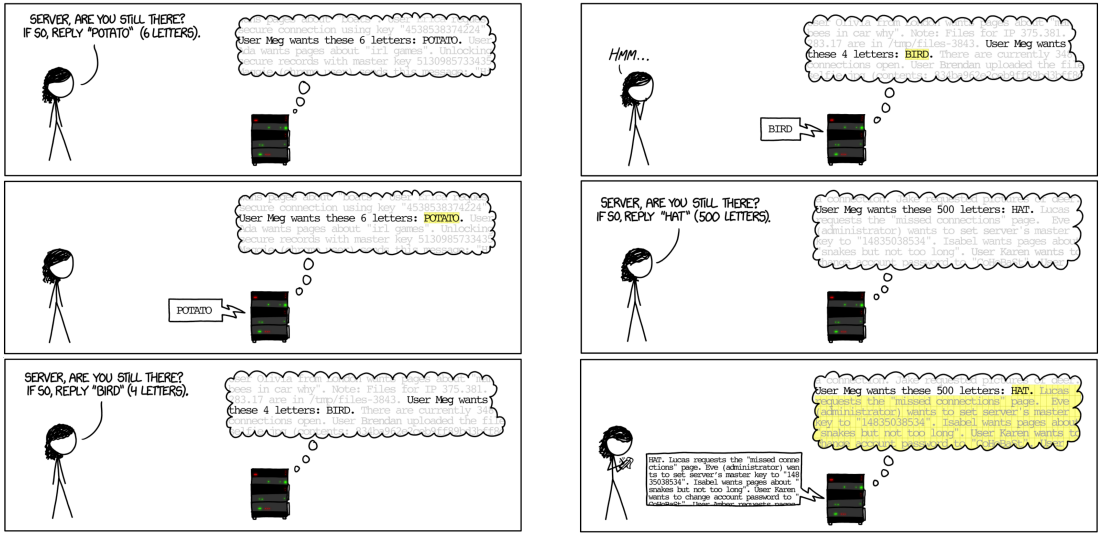
Problem?
over-write

```
1  void cat(char *dst, size_t n,
2    char *src1, size_t n1,
3    char *src2, size_t n2)
4  {
5    if (n1+n2 <= n) {
6      strncpy(dst, src1, n);
7      strncat(dst, src2, n-n1);
8    }
9  }
10 ...
```

Problem?
over-write

A famous over-read in OpenSSL

Heartbleed bug



https://xkcd.com/1354/

```c
void verify_stack()
{
  int verified = 0;
  char buf[8];
  gets(buf);
  /* <verification goes here> */
  if (verified) {
    printf("accept\n");
  } else {
    printf("reject\n");
  }
}
```

- Input 1: Neal ➠ reject
- Input 2: Caffrey ➠ reject
- Input 3: Overflow! ➠ accept

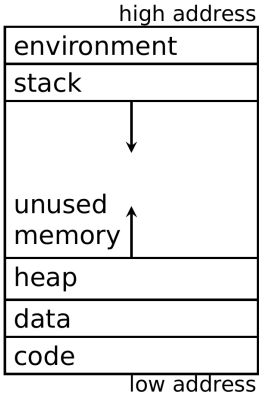
Buffer overflow occurs when writing outside of a buffer's boundaries

```c
void verify_heap()
{
  char *buf = malloc(8*sizeof(char));
  int *verified = malloc(sizeof(int));
  *verified = 0;
  gets(buf);
  /* <verification goes here> */
  if (*verified) {
    printf("accept\n");
  } else {
    printf("reject\n");
  }
}
```

- Input 1: Neal ➠ reject
- Input 2: Caffrey ➠ reject
- Input 3: aaa...aaa ➠ accept

Memory layout

- Heap grows towards higher addresses
  - Manual memory (de)allocation
- Stack grows towards lower addresses
  - Automatic memory (de)allocation
  - Each function has a 'stack frame'
- Data: e.g., global and static variables
- Code: instructions that CPU can process



Stack frames

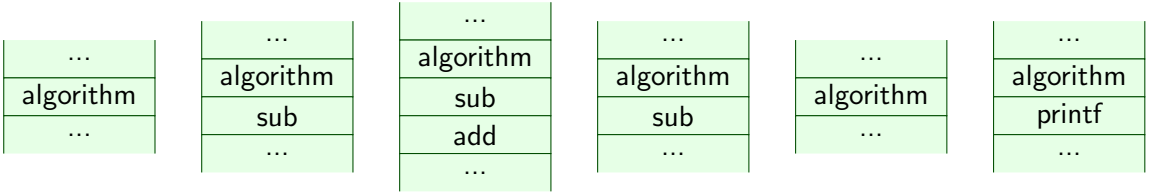
```
1  int add(int a, int b)
2  {
3    int result;
4    result = a+b;
5    return result;
6  }
7
8  int sub(int a, int b)
9  {
10   int result;
11   result = add(a,-b);
12   return result;
13 }
14
15 int algorithm()
16 {
17   printf("result: %d\n", sub(2,1));
18 }
```

Each function gets its own stack frame

- Local variables
- Function parameters
- Housekeeping such as:
  - Return address
  - Register values

Push ordering ➠ see calling conventions

Pushing an popping stack frames



Return to caller's context using housekeeping information

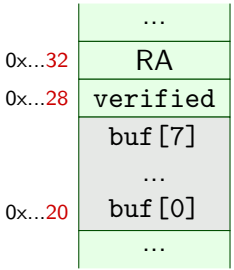
Buffer overflow on the stack continued (1/2)

```
1  void verify_stack()
2  {
3    int verified = 0;
4    char buf[8];
5    gets(buf);
6    /* <verification goes here> */
7    if (verified) {
8      printf("accept\n");
9    } else {
10     printf("reject\n");
11   }
12 }
```

- Input 4: aaa...aaa ➠ segfault
- Why not segfault on heap?
- Why segfault on stack?

Buffer overflow on the stack continued (2/2)

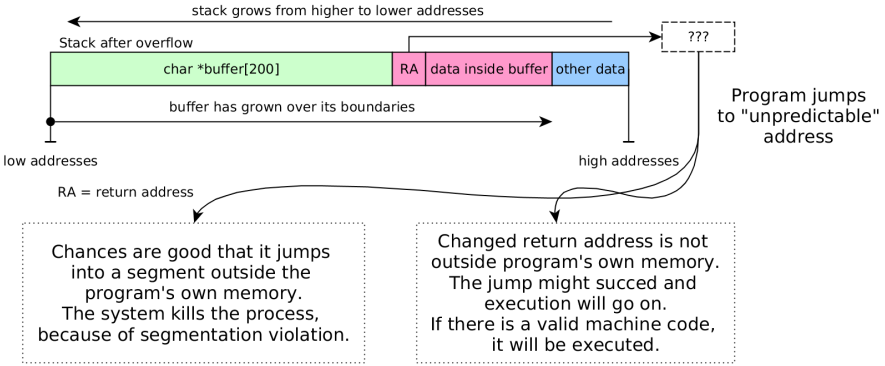
```
1  void verify_stack()
2  {
3    int verified = 0;
4    char buf[8];
5    gets(buf);
6    /* <verification goes here> */
7    if (verified) {
8      printf("accept\n");
9    } else {
10     printf("reject\n");
11   }
12 }
```



Can we solve the problem by pushing `buf` first?

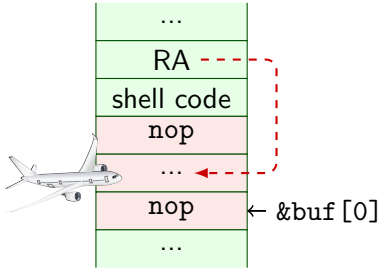
Buffer overflow that leads to code execution

Summary of principles for stack smashing attacks



1. Gain control of return address
2. Point to some malicious code

1. Add asm instructions in the buffer
   - Usually to open a shell
   - 'Shell-code'
2. Jump to the buffer's shell-code



Exact address of buffer?

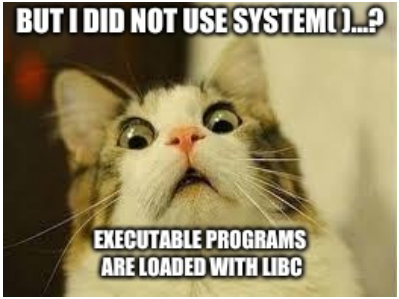
⇓

Use a nop-sled

Note: nop-sled + asm may also be injected to the heap—'heap spraying'

1. Point to an already loaded function
   - `system()`
2. Prepare stack with arguments
   - `"rm -rf /home/$USER"`

➠`system("rm -rf /home/$USER")`




```
$ cat main.c
int main() { return 0; }
$ gcc main.c
$ ldd ./a.out
    linux-vdso.so.1 (0x00007fff3a9e4000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa5bfeda000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fa5c04cd000)
```

Widely used stack smashing mitigation techniques

Idea: try to prevent the two necessary stack smashing conditions from meeting met

- Address randomization — Increases jump uncertainty
- Non-executable memory — Stop if instruction pointer gets here
- Stack canaries — Stop if RA got tampered with

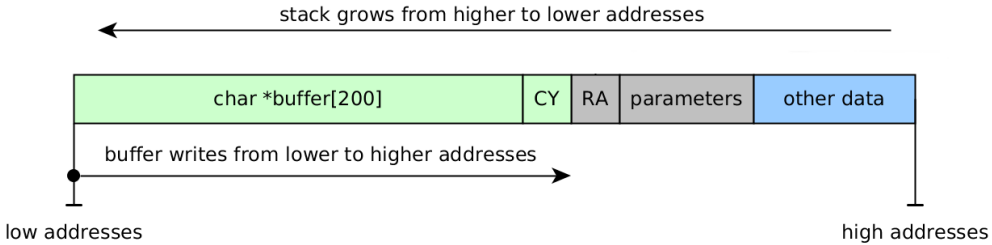
```
$ ldd ./a.out
  linux-vdso.so.1 (0x00007ffdda7ce000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7f307ba000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f7f30dad000)
$ ldd ./a.out
  linux-vdso.so.1 (0x00007ffe387d4000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fdd793ef000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fdd799e2000)
```

Canaries

"[...] miners would bring a caged canary into new coal seams. Canaries are especially sensitive to methane and carbon monoxide [...], as long as the bird kept singing, the miners knew their air supply was safe."

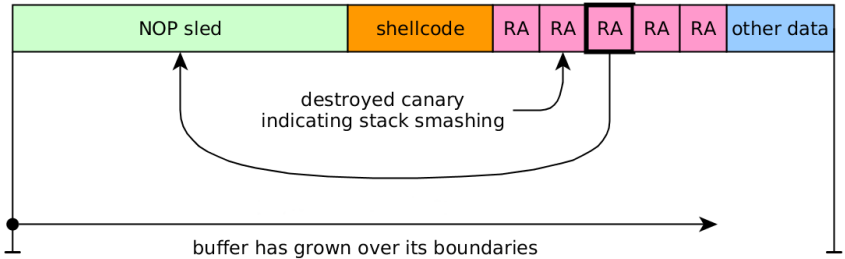
"**Short but meaningful**"

https://www.wisegeek.com/
what-does-it-mean-to-be-a-canary-in-a-coal-mine.htm



- Terminator canaries
- Random canaries
- ...

Require:
CY must be valid to use RA



Stop running—invalid canary value!

Can anyone think of examples where the two canary types fail?

Great mitigation techniques, but we realize that...

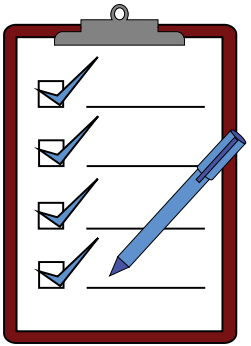
it is still imperfect

What else can we do?

- Avoid bugs in C/C++ code
- Build and use tools that help catching bugs
- Use memory safe programming languages

Avoid bugs in C/C++ code

- Sanitize all untrusted user input
- Manually verify all bounds correctly
- Be aware of integer underflow/overflow
- Use safe(r) functions and learn caveats
  - `fgets` vs. `gets`
  - `strncpy` vs. `strcpy`
  - `man strncpy` ➠ null-termination?
- **Adopt a secure coding standard**
  - CERT C[6]
  - MISRA C[7](embedded systems)



[6] https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf

[7] https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx

- Automated source code analysis before runtime
- Output warnings if errors are suspected
- Assess compliance with coding standards

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

int main()
{
  char *buf = malloc(8);
  fgets(buf, 8, stdin);
  printf("%s\n", buf);
  return 0;
}
```

```
$ splint main.c
```

➠ 3 non-gcc warnings

Details: https://www.splint.org/

[8] https://www.perforce.com/blog/qac/what-static-code-analysis

- Analyze what program does at runtime
- Valgrind—look for memory errors
- Fuzzing[9]—what happens on funky input?
  - Random and mutation-based
  - Structure-aware
  - Program-aware
  - Automated feedback loops
  - ...

https://upload.wikimedia.org/wikipedia/commons/0/08/Rabbit_american_fuzzy_lop_buck_white.jpg

[9] Brief introduction and a few demos: https://www.youtube.com/watch?v=dMmsPwkSqOc

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg 1	libjpeg-turbo 1 2	libpng 1
libtiff 1 2 3 4 5	mozjpeg 1	PHP 1 2 3 4 5 6 7 8
Mozilla Firefox 1 2 3 4	Internet Explorer 1 2 3 4	Apple Safari 1
Adobe Flash / PCRE 1 2 3 4 5 6 7	sqlite 1 2 3 4...	OpenSSL 1 2 3 4 5 6 7
LibreOffice 1 2 3 4	poppler 1 2...	freetype 1 2
GnuTLS 1	GnuPG 1 2 3 4	OpenSSH 1 2 3 4 5
PuTTY 1 2	ntpd 1 2	nginx 1 2 3

`http://lcamtuf.coredump.cx/afl/`

And around 120 more projects!

Use memory-safe programming languages

Intuition: `x[y] = z` should stop normal program execution if `x` is non-array or `y` is out-of-range, and you should not operate on raw memory addresses[10]

Challenges:

- You need low-level access to hardware
- You inherit a large `C/C++` project
- Someone must implement the core correctly

GOLANG 

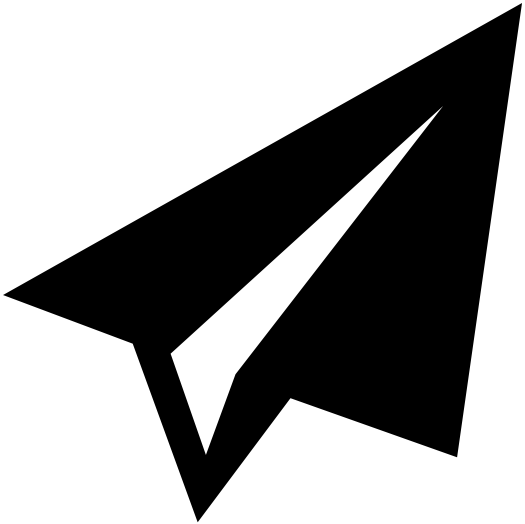
C#

What about performance? Is it a valid concern?

[10] If you want a more precise intuition: `http://www.pl-enthusiast.net/2014/07/21/memory-safety/`

Summary

- Weird machines
- Over-read, over-write
- Change program flow
- Mitigation techniques
- Tools and other options



Assignment questions (1/3)

1. Suppose that the code below is compiled as follows: `gcc -Wall -Werror -std=c99 main.c`. Provide two integer inputs that will result in 'unintended behaviour' and name what this threat is called. Make assumptions if necessary.

```c
#include <stdio.h>

int get_int() {
  int v; printf("Enter an integer: ");
  scanf("%d", &v);
  return v;
}

int main() {
  int a=get_int(), b=get_int(), max=10;
  if (a+b > max)
    printf("%d+%d > %d\n", a, b, max);
  else
    printf("%d+%d <= %d\n", a, b, max);
}
```

2. Determine which compiler option could be used to ensure that the program aborts if such unintended behaviour occurs. Does this solution work for `unsigned ints`? Why (not)?

3. What is the compliant way of adding two unsigned integers according to CERT C standard?

4. Suppose that the code below is compiled as follows: `gcc -Wall -Werror -std=c99 -fno-stack-protector main.c`. Explain the steps necessary to trigger the print statement. Make assumptions if necessary.

```c
#include <stdio.h>
#include <limits.h>
#define SECRET UINT_MAX

void gotcha() { printf("Gotcha!\n"); }

int main() {
  unsigned secret = 0;
  char buf[8];
  scanf("%s", buf);
  if (secret == SECRET) {
    gotcha();
  }
  return 0;
}
```

5. How would you adapt your strategy if `SECRET` was set to `0xff0a0dff`? Explain principles.

6. Which type of buffer overflow mitigation technique does the new secret value remind you of?

7. Explain two other mitigation techniques that make it harder to execute code in a buffer overflow.

8. Attackers may use `nop`-sleds to increase the likelihood of jumping to their shell-code. To defend against this a colleague of yours suggested that all user input be filtered for repeated `nop` instructions. How would you trivially bypass such a filtering mechanism?

9. What is the difference between static and dynamic code analysis?

10. Briefly explain the process of fuzzing a program: how does it work and what is the goal? Name one fuzzer that found a buffer overflow vulnerability in a TLS library.

11. Suppose that you are hired by a consultant company to work on a brand new project. Explain the circumstances in which you would choose to program in `C/C++`, and why you might choose a different programming language in most other cases.

Any questions?