Systems 3

Vulnerabilities

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Photo by Krzysztof Niewolny on Unsplash

Photo by Kate Stone Matheson on Unsplash

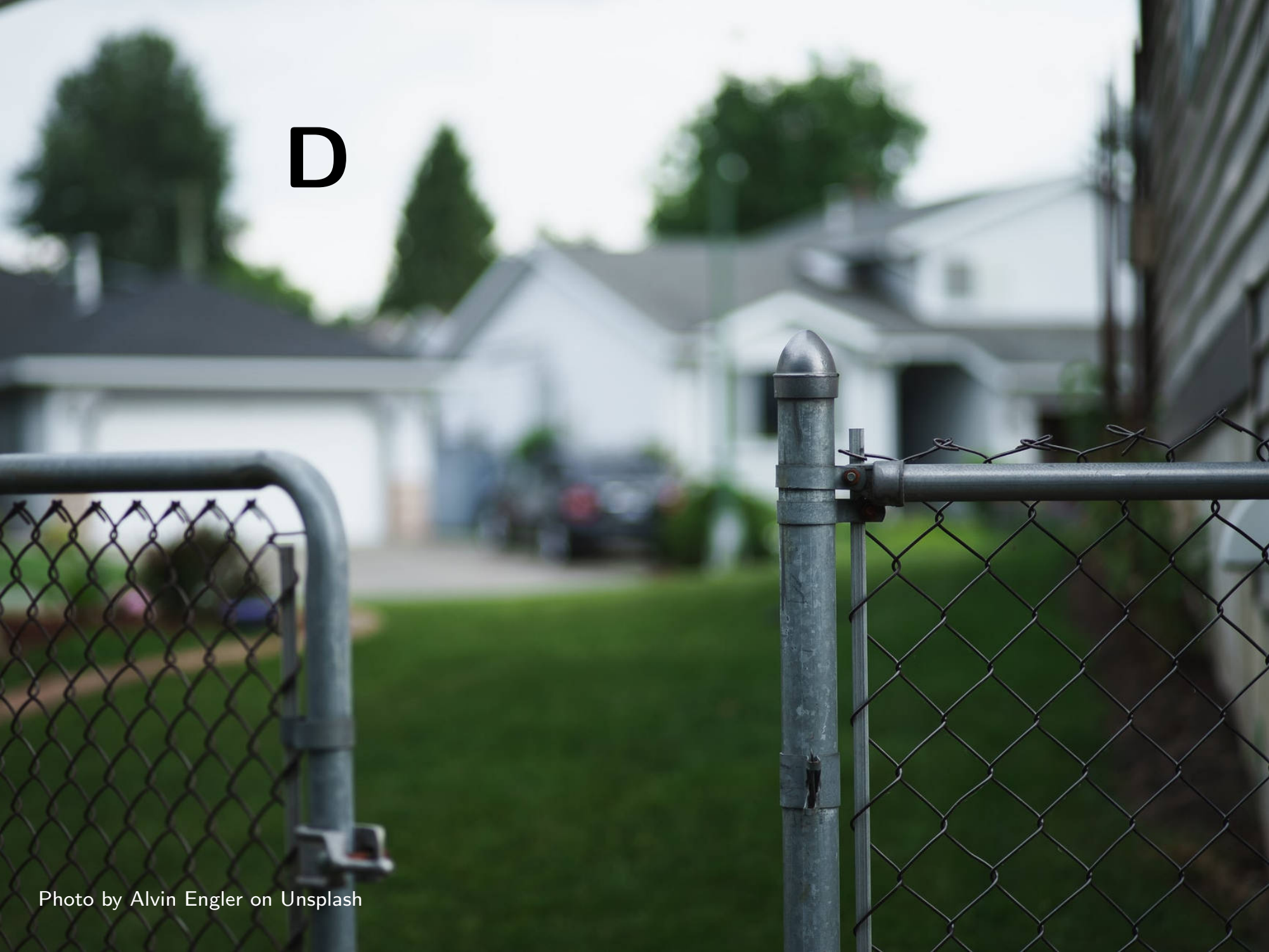


Photo by Alvin Engler on Unsplash

Chapter Goals

- Understand where security problems can come from.
- Understand typical attack mechanisms (overflow, injection, race)
- Understand what the OS (and compiler and runtime) are already doing to prevent them
- Understand the risks of writing code
- Be able to apply what remains to be done by the developer

Responsible Disclosure

1. Create a report
2. Contact company
   - security@, abuse@, noc@ (RFC2142[1])
   - bug bounty program
   - security.txt[2]
3. Wait for response and fix
4. Publish details
   - Google's disclosure policy[3]
   - Responsible Vulnerability Disclosure Process[4]

[1] https://www.ietf.org/rfc/rfc2142.html

[2] https://securitytxt.org/

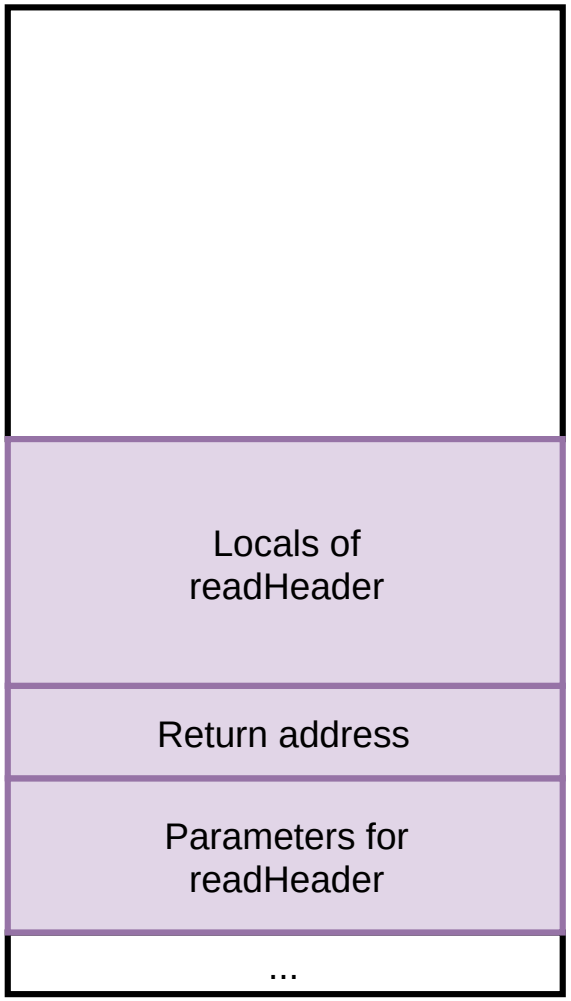
[3] https://googleprojectzero.blogspot.com/2015/02/feedback-and-data-driven-updates-to.html

[4] https://tools.ietf.org/html/draft-christey-wysopal-vuln-disclosure-00

Call stack revisited

Call stack is used to store

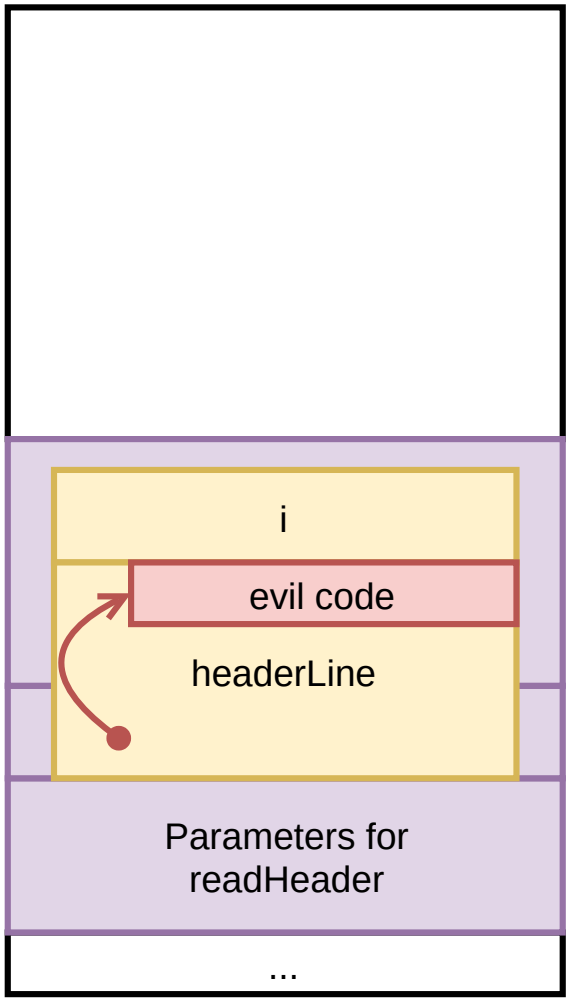
- Local variables
- Parameters
- Return addresses



Simple Example

```
1  int readHeader(const char *input) {
2     int i = 0;
3     char headerLine[256];
4
5     while (input[i] != '\n') {
6        headerLine[i] = input[i];
7        i++;
8     }
9     return i;
10 }
```

Question: Does an attacker have to know the exact address of his code?
Answer: No. Usually a nop slide is prepended.
Notice: Also applicable to heap (heap spraying).



Stack Canaries

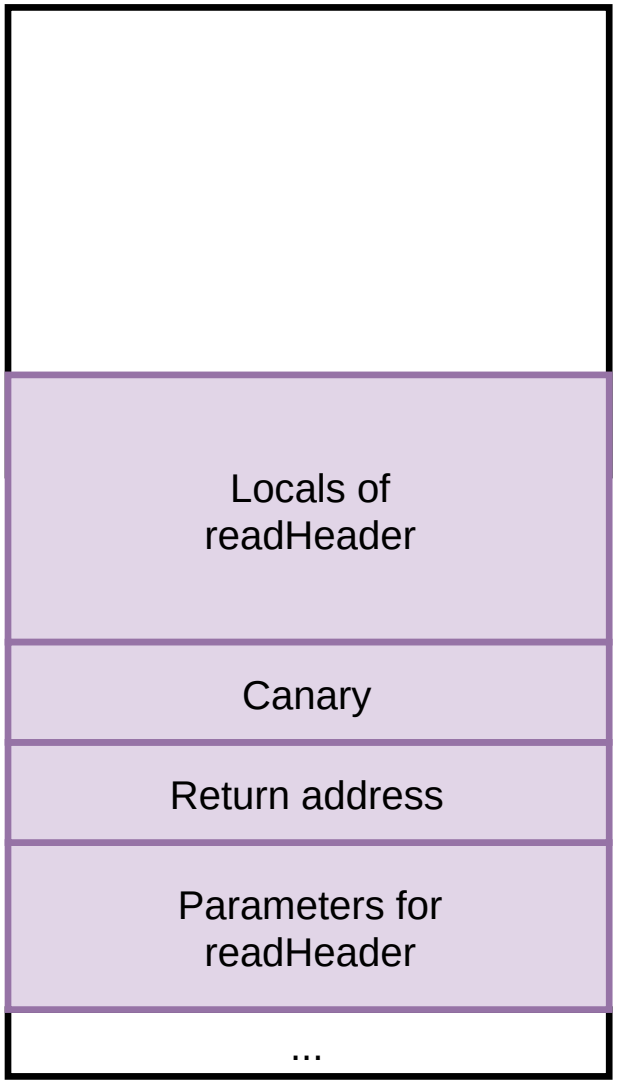
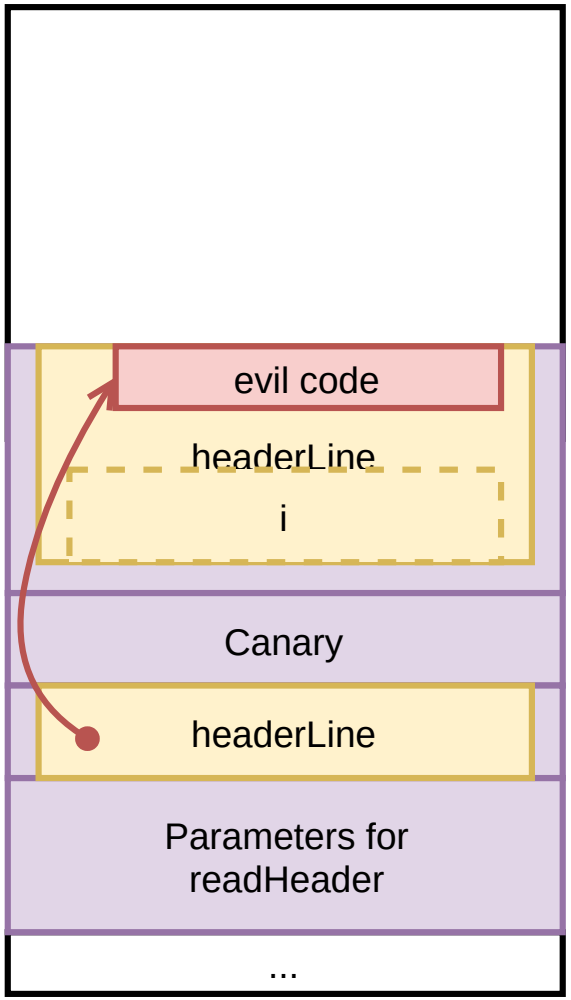


Photo by Kaikara Dharma on Unsplash

Avoiding Stack Canaries

```
1  int readHeader(const char *input) {
2    char headerLine[256];
3    int i = 0;
4
5    while (input[i] != '\n') {
6      headerLine[i] = input[i];
7      i++;
8    }
9    return i;
10 }
```



See also https://security.stackexchange.com/questions/20497/

Data Execution Prevention (DEP)

A modern CPU can distinguish between data and text segments (NX bit[5]).

- heap, stack and global variables are writable, but not executable
- code is executable, but not writable

This policy is also known as W XOR X in the OpenBSD community.

[5] https://en.wikipedia.org/wiki/NX_bit

Attacking DEP

Why inject code, when there is already plenty of it there?

Practical example: http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

Return to libc

- libc contains common functionality
- libc is linked basically with every program
- An attacker could open a new shell via `system`
- Many similar attacks possible

Return-oriented programming

Basic idea: Concatenate existing instructions to a malicious program.

1. Look for small sequences of code (gadgets)
2. Build a stack that "jumps" to one gadget after the other[6]

There are tools like ROPgadget[7] which help you find specific gadgets.

Question: Can every piece (line) of code be used as gadget?
Answer: Yes if it's useful, but it's executed until the next return statement. So usually a gadget consists of a few instructions before the next jump.

[6] Using return operations
[7] http://shell-storm.org/project/ROPgadget/

Address-Space Layout Randomization (ASLR)

Return-oriented programming only works if the attacker knows the right address. ASLR randomizes the positions of the initial stack, the heap, and/or the libraries to complicate attacks.

There are a couple of issues with ASLR:

- many implementations have code at fixed locations (at least some)
- entropy is too low
- offset of functions in library is always the same

Question: What is the big advantage of ASLR to other systems?
Answer: While complicating attacks the performance is not affected.

Attacking data

```
1  void printGrades(const char *username, const char *password) {
2     int isAuthorized = 0;
3     char studentNumber[128];
4
5     isAuthorized = checkCredentials(username, password);
6
7     printf("Enter student number: ");
8
9     gets(studentNumber);
10
11    if (isAuthorized) {
12       // get and print grades
13    } else {
14       printf("Credentials invalid");
15    }
16 }
```

Question: Who can spot the issue?

printf(3) is dangerous

```
SYNOPSIS
        #include <stdio.h>

        int printf(const char *format, ...);

DESCRIPTION
   Conversion specifiers
       n      The number of characters written so far is stored into the
              integer pointed to by the corresponding argument.  That argument
              shall be an int *,  or variant whose size matches the
              (optionally) supplied integer length modifier.  No argument is
              converted. The behavior is undefined if the conversion
              specification includes any flags, a field width, or a precision.
```

Question: Any idea why printf can be dangerous?
Answer: Variable number of arguments and printf can modify data.

Bad `printf` example

- User can inject format
- User can read the stack (`%08x`)
- User can write on the stack (`%n`)

```
1 char greeting[] = "Hello ";
2 char name[128];
3
4 gets(name);
5
6 strcat(greeting, name);
7
8 printf(greeting);
```

Further information:
https://www.owasp.org/index.php/Format_string_attack or
https://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf

Are you brave enough to click on those links? ;-)

Using invalid memory

```
1  int *request = malloc(4096 * sizeof(char));
2
3  int size = readRequest(request);
4
5  if (size == 0) {
6     printf("Received empty request");
7
8     free(request);
9  }
10
11 ...
12 ... A lot of code
13 ...
14
15 getc(request);  /* maybe modify different data structure */
```

Nice article (even if published in 2007): Heap Feng Shui in Javascript (http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html or https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf)

Attacking the null pointer

This attack is not possible with a modern kernel, but there are still old kernels in the wild and it's a nice example. A Null Pointer Dereference Attack works like this:

1. put code at address zero (`mmap`)
2. let the kernel dereference a null pointer

Question: Are user and kernel space not separated?
Answer: To get a better performance, the kernel space is often mapped into the user space. (See also Meltdown+Spectre, later.)

Code injection...

...is a common issue in every program language (e.g. SQL injection, JS injection, shell injection, ...).

The basic idea is that the user is able to manipulate a command, like in the following example:

```
1 char cmd[256] = "echo Hello ";
2 char *name = NULL;
3
4 printf("Enter your name: ");
5 scanf("%ms", &name);
6
7 strcat(cmd, name);
8 system(cmd);
9 free(name);
```

User could enter something like `foo; rm -rf /`.

Note: '%ms' does a right-sized `malloc(3)` (cool!). See `scanf(3)`.

Exploiting race conditions

Exploitable code:

```
1  int fd;
2
3  /* Using real UID */
4  if (access("./.well-known/CylmEesyudneyd1", W_OK) != 0) {
5     exit(1);
6  }
7
8  /* Malicious program could remove file and create a symbolic link */
9
10 /* Using effective UID */
11 fd = open("./.well-known/CylmEesyudneyd1", O_WRONLY);
12 write(fd, someInput, sizeof(someInput));
```

From access(2): the use of this system call should be avoided.

Literature

- Modern Operating Systems, *Andrew S. Tanenbaum, Herbert Bos*, Fourth Edition, Chapter 9.7