Network Security

AA 2015/2016

Vulnerabilities

Dr. Luca Allodi

Software bugs

- A bug is a problem in the execution of the software that leads to unexpected behaviour
  - Software crashes
  - Wrong entries are displayed/stored in a backend database
  - Execution loops infinitely
  - ..
- Characteristics of a bug
  - Replicability
  - Logic/configuration/design/implementation
  - Fix priority
  - If it's documented, it's a feature

An example of a sw bug (pseudocode)

```
int login(database, context){
        char username[10];
        char password[10];
        printf("login:"); gets(username);
        printf("password:"); gets(password);
        correct_pwd=lookup(username, database);
        if (correct_pwd!=password)
                printf('Login failed');
                return;
        else{
                printf('login succeeded');
                exec(context);
        }
        return 1;
}
```

Swap it around..

```
int login(database,context){
        char username[10];
        char password[10];
        printf("login:"); gets(username);
        printf("password:"); gets(password);
        correct_pwd=lookup(username, database);
        if (correct_pwd==password)
                printf('Login succeeded');
                exec(context);
        else{
                printf('Login failed');
                return;
        }
        return 1;
}
```

Vulnerabilities

- *A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy*

Definition from NIST SP 800-30

Types of vulnerabilities

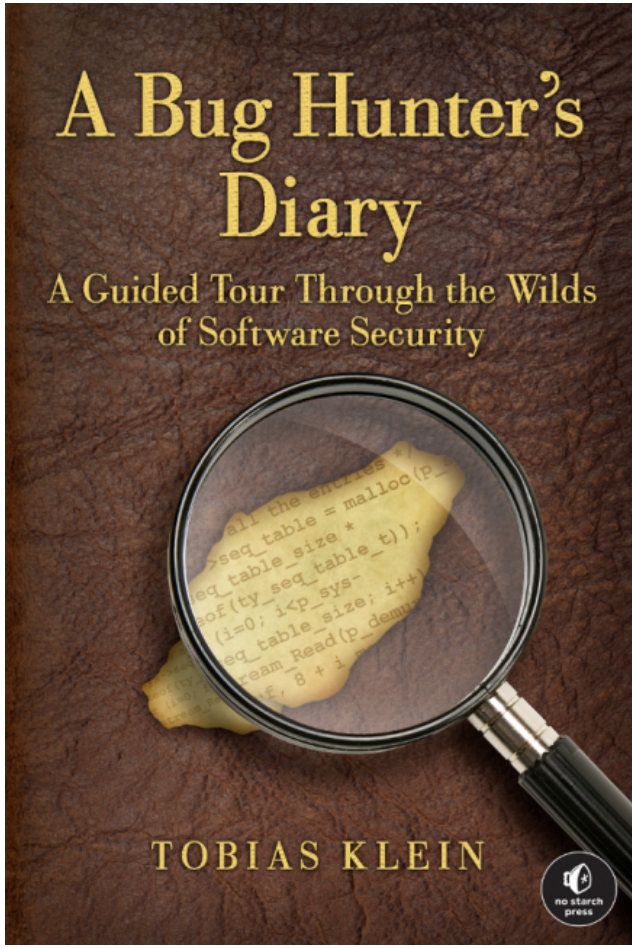
- Vulnerabilities can be found at any level in an information system
  - Configuration vulnerabilities
  - Infrastructural vulnerabilities
  - Software vulnerabilities
- Configuration vulnerabilities
  - Software or system configuration does not correctly implement security policy
    - e.g. accept SSH root connections from any IP
- Infrastructural vulnerabilities
  - Design or implementation problems that directly or indirectly affect the security of a system
    - e.g. a sensitive database in a network's DMZ
- Software vulnerabilities
  - Design or implementation of a software module can be exploited to bypass security policy
    - e.g. authorisation mechanism can be bypassed

Software vulnerabilities

- Our focus in this course
- Thousands of software vulnerabilities are discovered each year
  - Some are publicly disclosed
  - Some are not
- MITRE → non-profit organisation (Massachusetts, U.S.A.)
  - Supports, among others, activities from
    - Department of Homeland Security (DHS)
    - Department of Defense (DoD)
    - National Institute for Standards and Technology (NIST)
  - Maintaines standard for vulnerability identification
    - Common Vulnerabilities and Exposures (CVE)

Vulnerability discovery

- Vulnerabilities are widely different in nature
  - Often implementation-dependent
  - May require deep understanding of sw module interaction
  - Necessary in-depth knowledge of system design
    - e.g. kernel structure, memory allocation,..
- Two main discovery techniques
  - Code lookups
    - Manual/semi-automatic search in codebase for known patterns
  - Fuzzing
    - Semi-automatic random input generation--> try to crash program
  - Bonus technique: "Google hacking"
    - Look for known vulnerable functions in google → returns vulnerable webpages



Vulnerability discovery and disclosure

- Can be found either internally or externally to a company
  - **Internally** → managed within the company
    - Patch (fixing) prioritisation
    - Communication to customers
  - **Externally** → found by an external security researcher
    - Disclosure to vendor
      - Payment
    - Patching prioritisation
    - Disclosure to public

Vulnerability handling

- Internal process must
  - Accept information about new vulnerabilities
    - Internal or external sources
  - Verify vulnerability report
  - If vulnerability exists
    - Develop resolution
    - Post-resolution activities
- ISO 30111

Vulnerability handling – verification phase

- Initial investigation
  1. The reported problem is a security vulnerability
     - Must have repercussions over security policy
  2. The vulnerability affects a supported version of the software the vendor maintains (e.g. not caused by 3rd party modules).
     - Else, exit process
  3. The vulnerability is exploitable with currently known techniques
     - Else, exit process
  4. Root cause analysis
     - Underlying causes of vulnerability and look for similar problems in the code
  5. Prioritisation
     - Evaluate potential threat posed by the vulnerability

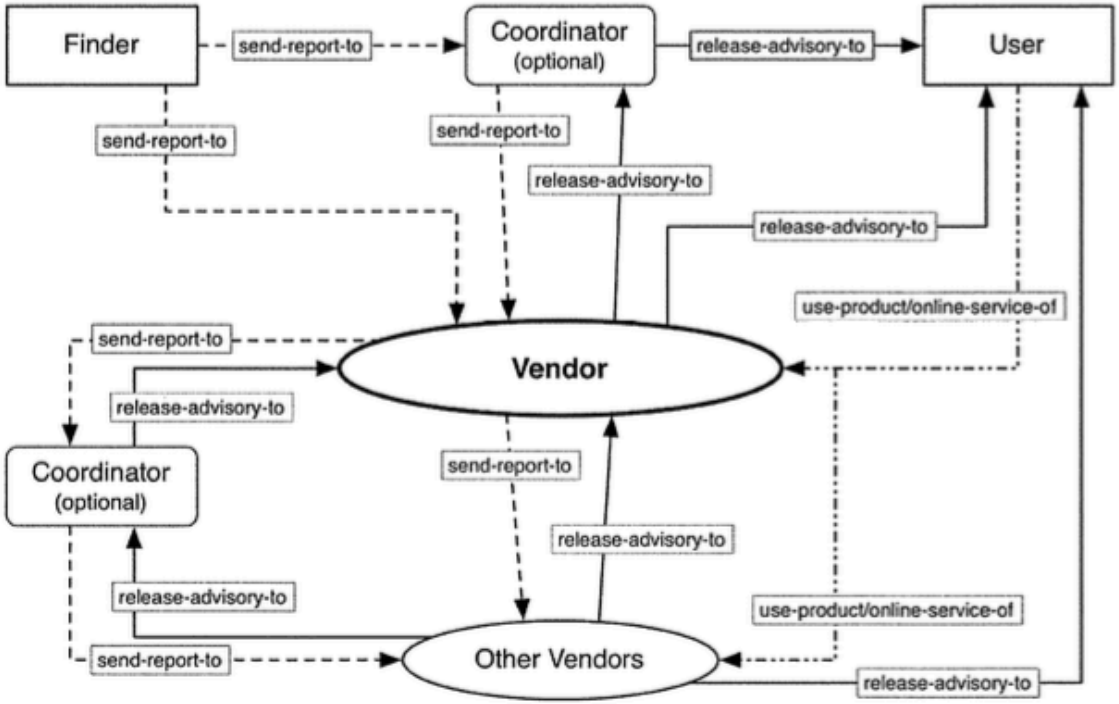
Vulnerability handling – resolution and release phases

- Resolution decision
  - Vendor must decide how to resolve the vulnerability
  - Different decisions for different types of vulnerabilities
    - Configuration vulnerabilities → advisory may be enough
    - Code vulnerabilities → patch
    - Critical vulnerabilities → release a mitigation before full patch
- Remediation development
  - Every resolution must be tested before being delivered to clients
    - minimize negative impacts caused by software change
- Release
  - Web services → vendor deploys patch itself
  - Stand-alone product → patch release (see ISO 29147)
- Post-release
  - Monitor situation (e.g. patch may not be always effective)
    - Support to final client

Vulnerability disclosure

- Vulnerabilities are information sets
- The vulnerability disclosure process is about information exchange – ISO 29147
  - Finder → vendor
  - Vendor → user

Picture from ISO 29147



Confidentiality of vulnerability information

- Vulnerability information is considered sensitive and confidential by vendors
  - Pose a threat to end users
  - May affect vendor's reputation
- Build secure communication channels to preserve confidentiality and integrity of information
- Vulnerability advisories are typically published after patching
  - Internal policies determine whether a vulnerability will be published or not
    - Typically a function of vulnerability severity

Issues with vulnerability disclosure – the case of external finders

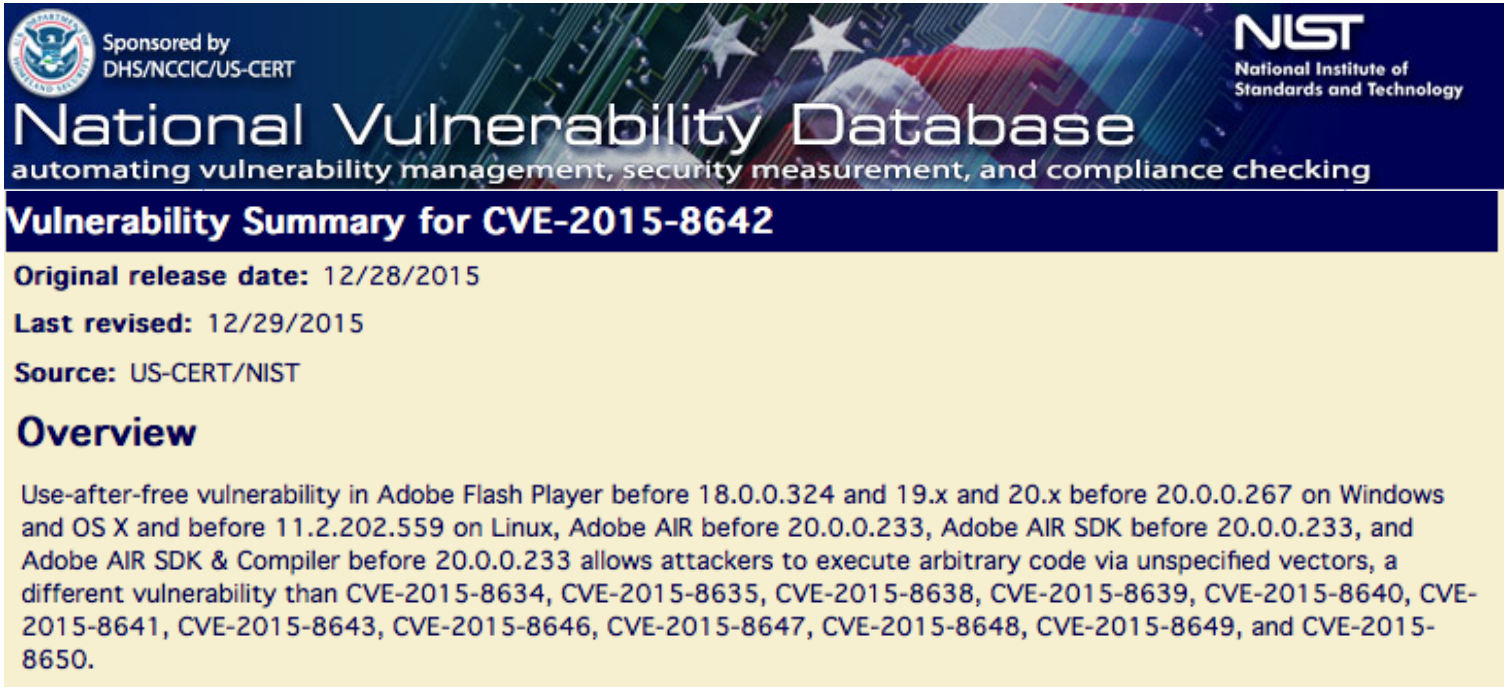
- Security researcher that finds vulnerability may expect
  - Economic return
  - Credit (to mention on curriculum)
- Issue → how to communicate vulnerability to vendor?
  - Say too little → vulnerability not reproducible → no $$$
  - Say too much → vulnerability fully known → thanks for the info → no $$$
- Agreement between sec researcher and vendor
  - Third party mediates (e.g. ZDI)
  - Bug bounty programs (e.g. Microsoft, Google)
  - Credit assured (e.g. Apple?)
- Often involves development of Proof-of-Concept exploit that shows the vulnerability is exploitable
- For more on vuln disclosure issues see "Miller, Charlie. "The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales." *In Sixth Workshop on the Economics of Information Security*. 2007."

Third party mediators

- Several on the market, act as proxy between security researcher and vendor
  - Communicate vulnerability to vendor
  - Hold vulnerability information for a certain amount of time (typically 60-90 days)
  - When hold period expires they disclose the vulnerability
    - Mechanism to push vendors to patch
  - Secunia, ZDI, SecurityFocus, …
- If vulnerability is known before vendor releases patch → "zero day vulnerability"
- Google Zero Day Project
  - Discover vulnerabilities (often in competitors' software)
  - Aggressively release vuln info after deadline expires

National vulnerability database

- NVD for short → NIST-maintained database of disclosed vulnerabilities
  - The "universe" of vulnerabilities



National Vulnerability Database (2)

External Source: CONFIRM

Name: https://helpx.adobe.com/security/products/flash-player/apsb16-01.html

Type: Advisory; Patch Information

Hyperlink: https://helpx.adobe.com/security/products/flash-player/apsb16-01.html

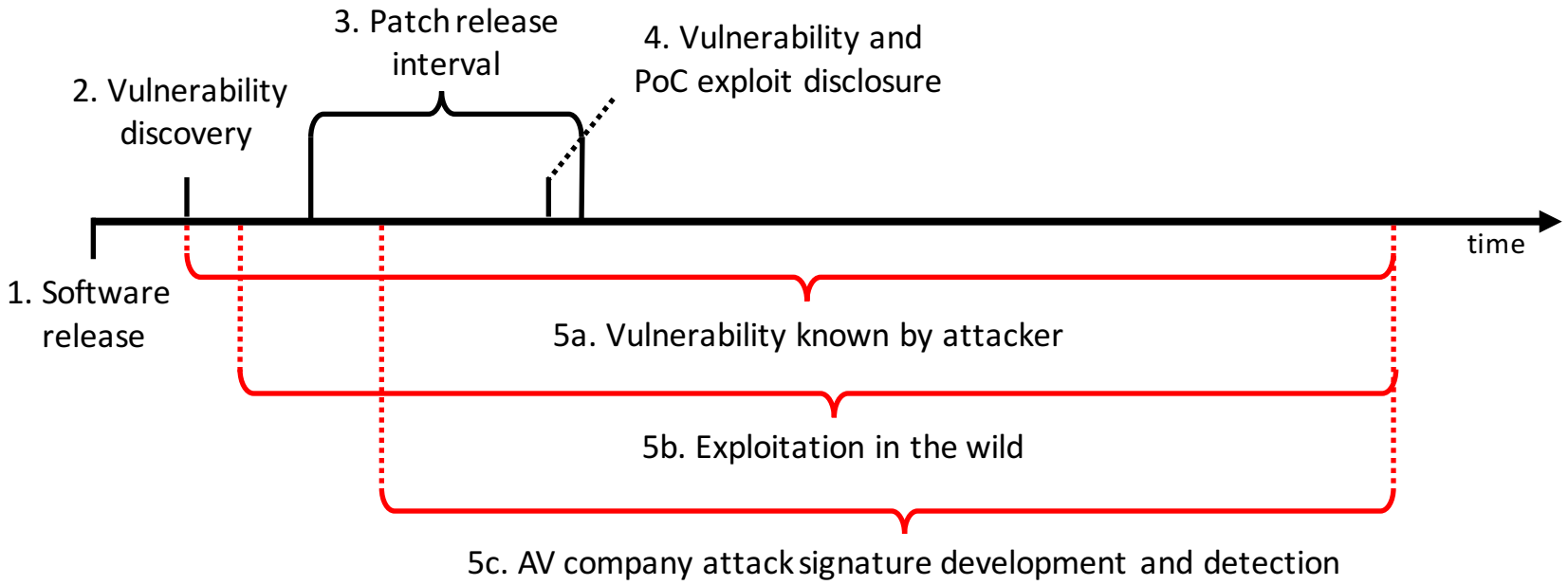
Vulnerable software and versions

+ Configuration 1
  + AND
    + OR
      * cpe:/a:adobe:air_sdk:20.0.0.204 and previous versions
      * cpe:/a:adobe:air_sdk_%26_compiler:20.0.0.204 and previous versions
    + OR
      cpe:/o:apple:mac_os_x
      cpe:/o:apple:iphone_os
      cpe:/o:google:android
      cpe:/o:microsoft:windows
+ Configuration 2
  + AND
    + OR
      * cpe:/a:adobe:flash_player:20.0.0.235
      * cpe:/a:adobe:flash_player:20.0.0.228
      * cpe:/a:adobe:flash_player:19.0.0.245
      * cpe:/a:adobe:flash_player:19.0.0.226
      * cpe:/a:adobe:flash_player:19.0.0.207
      * cpe:/a:adobe:flash_player:19.0.0.185
      * cpe:/a:adobe:flash_player:18.0.0.268 and previous versions
    + OR
      cpe:/o:apple:mac_os_x
      cpe:/o:microsoft:windows

Vulnerability feeds

- Vulnerabilities are disclosed by publication in the NVD and other vulnerability feeds
  - Public and private
- Private feeds release information earlier
  - “early advisories”
  - Secuina, SecurityFocus, ZDI
- Public feeds typically release weekly or monthly updates
  - SANS@RISK
  - https://www.sans.org/newsletters/at-risk

Vulnerability life-cycle overview



Types of vulnerabilities

- Different types of vulnerabilities
- "The Open Web Application Security Project (OWASP) is a 501(c)(3) worldwide not-for-profit charitable organization focused on improving the security of software"
  - https://www.owasp.org/index.php/Main_Page
- Good resource for information security resources
- "Top 10 vulnerability threats"
  - Good overview of most common vulnerability types with examples

Injection vulnerabilities

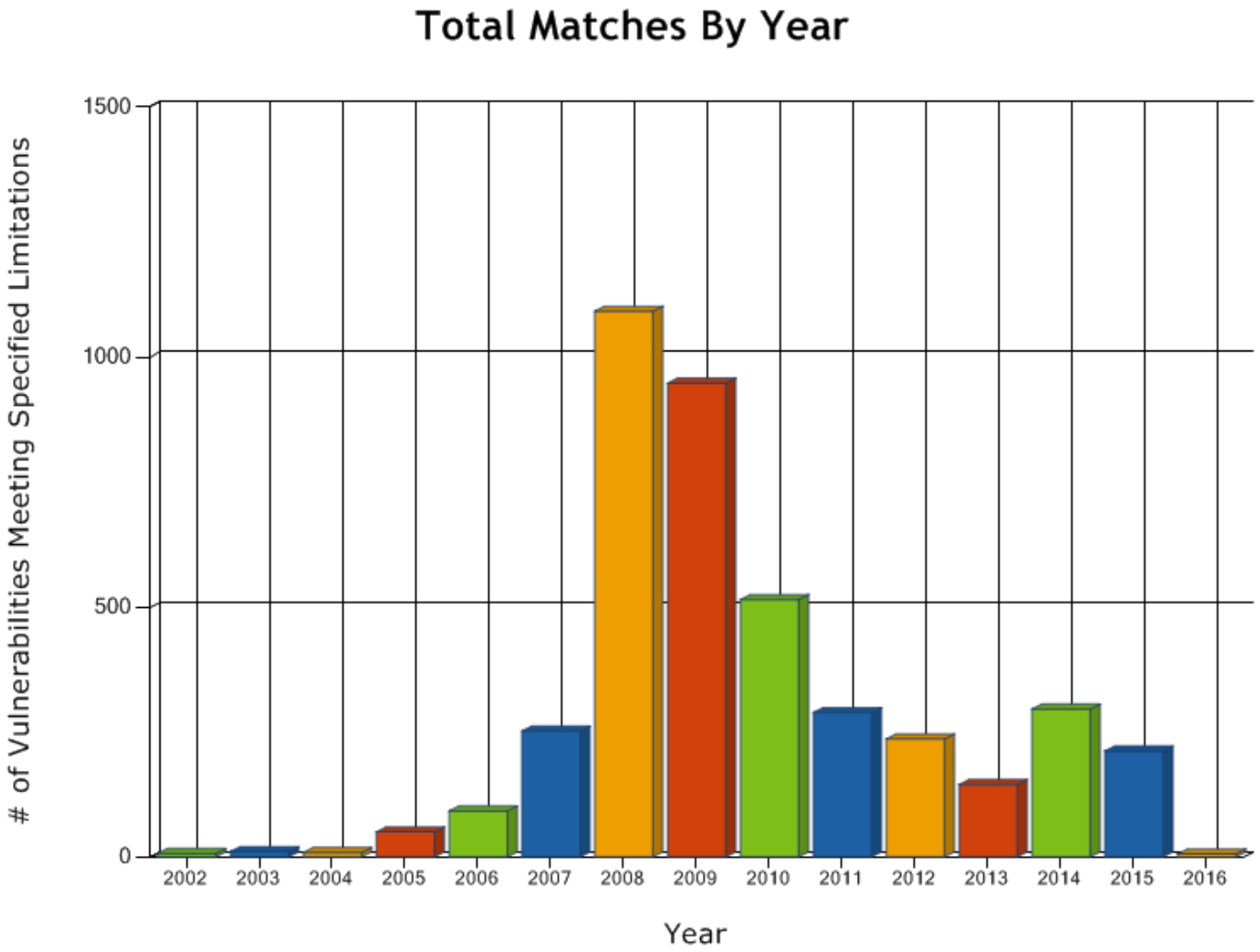
- Possibly the most common type of vulnerabilities
  - Exploits unsecure or not robust input channels to applications
- Input to the application can be forged in such a way that the application (or application backend) executes some commands
- Example:
  - SQL injection → inject SQL queries through an interface (typically web) by inputting malicious strings
    - String is interpreted by MySQL server as a query
  - Buffer Overflow

SQL Injection example

- Imagine a website with an input field "username"
  - The user inputs their name and the backend returns all their details
- The query on the backend will look something like this:
  - SELECT * from USERS where name='$user'
  - Where $user is the value set in the input username above, interpreted as a string
- The attacker can set $user=superpippo' OR 'owned'='owned
  - The backend will then interpret the following query

  → SELECT * from USERS where name='superpippo' OR 'owned'='owned'
- That's a valid query that returns all fields in USERS
- Mitigation → input validation (e.g. do not allow special characters in input fields).

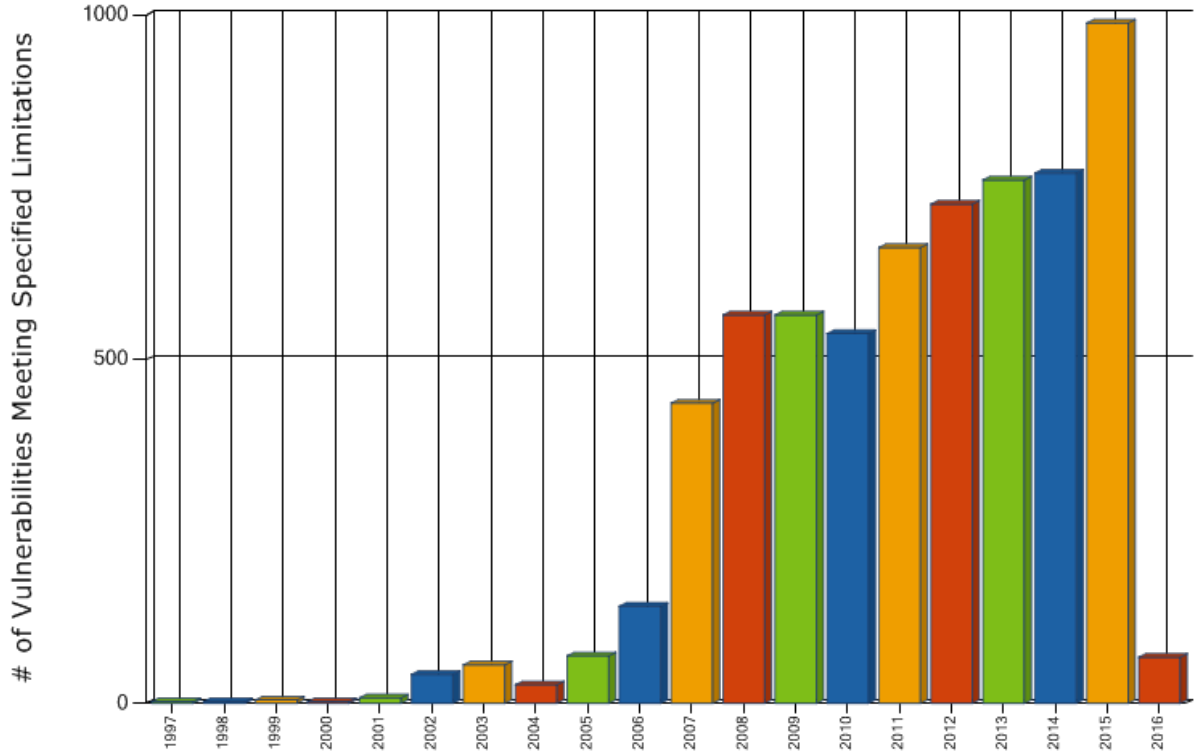
SQL Injection vulnerabilities



Stats from NVD (Feb 2016)

Buffer overflows

- May happen when input is not properly validated
- Input overwrites memory in such a way that execution can be controlled by the attacker
- Very common types of vulnerabilities
  - Extremely powerful as they typically allow the attacker to execute arbitrary code on the attacked system



Stats from NVD (Feb 2016)

Memory buffers – background notions

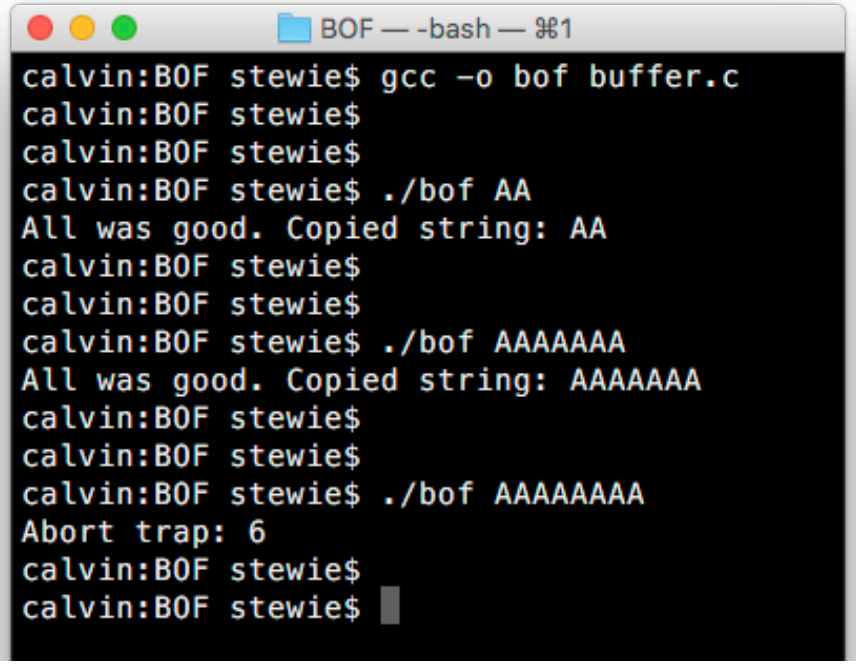
- Buffer → a block of memory that contains one or more instances of some data
  - Typically associated to an array (e.g. C, Javascript)
  - Buffers have pre-defined dimensions
    - Can accommodate up to x bytes of data
- Buffer overflow → the **input data dimension** exceeds the size of the buffer
  - Some input data "overflows" the buffer

Buggy code - example

buffer.c

```
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
int overflowme(char *string){
        char buffer[8];
        strcpy(buffer, string);
        printf("All was good. Copied
string: %s\n", buffer);
        return 1;
}

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



Trap 6 = SIGABRT → signals the process to abort

Memory layout and CPU registers

Memory

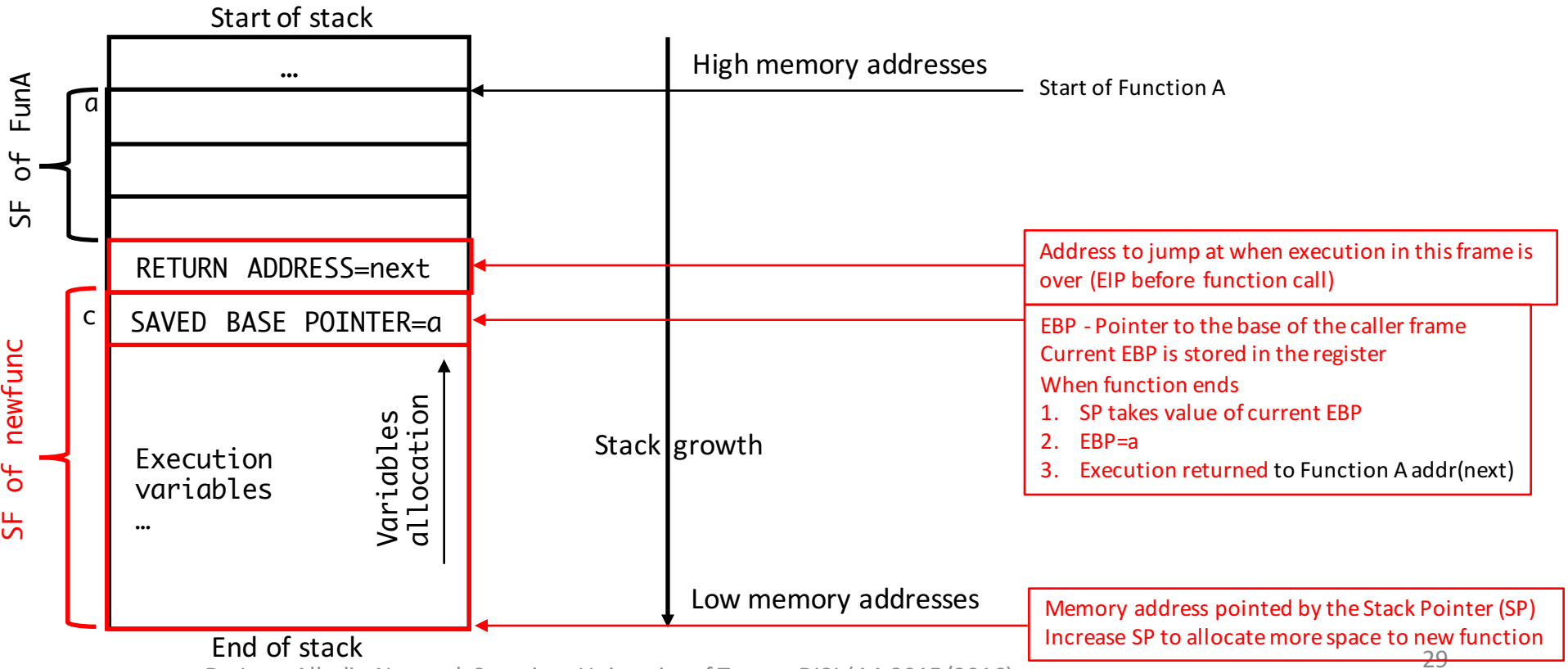
- Data + Text
  - The Data part references information on variables defined at compile-time
  - Text is the executable code of program
- Stack
  - Stores temporary information in memory
    - e.g. data set by called functions
  - LIFO → last-in-first-out
    - New "stack frames" are appended at the end of the current stack
  - Stack grows toward lower memory addresses
  - Stores RETurn address to go to when subroutine is over
- Heap
  - Data allocated run-time (malloc(), etc..)
  - Heap grows towards higher memory addresses

CPU registers

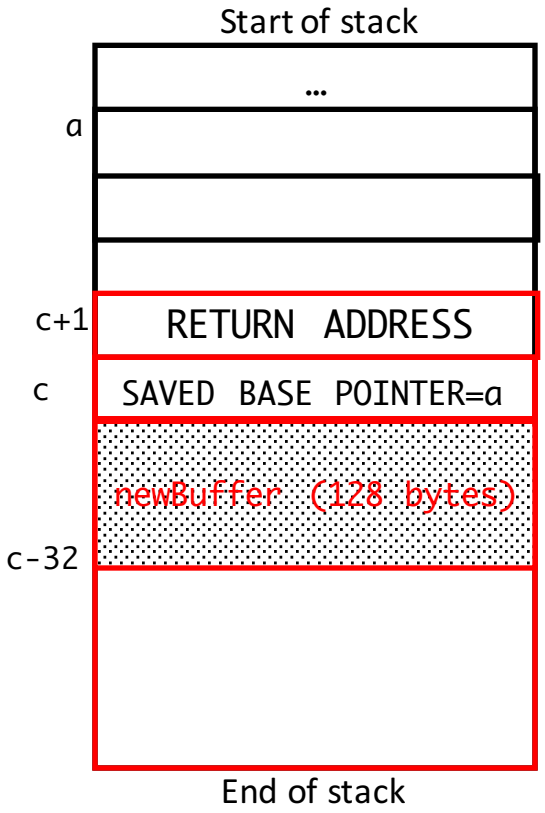
- Other information is stored in CPU registers
  - Depends on architecture
- x86 has several registers
- Here we are interested mainly in *pointer registers*
  - They point to areas of memory the execution will jump to
- EBP→ stack base pointer
  - Address of current stack frame
- SP → stack pointer
  - Address to end of stack
- EIP → instruction pointer
  - (offset) memory address of next instruction to be executes
  - EIP at subroutine call → RET

Buffer overflow – background (x86 32 bits)

- When called, functions are "appended" to the memory stack
  - a new "stack frame" is created
- Buffers are areas of memory that are allocated to store (input) data



Buffer overflow – attack (x86 32 bits)

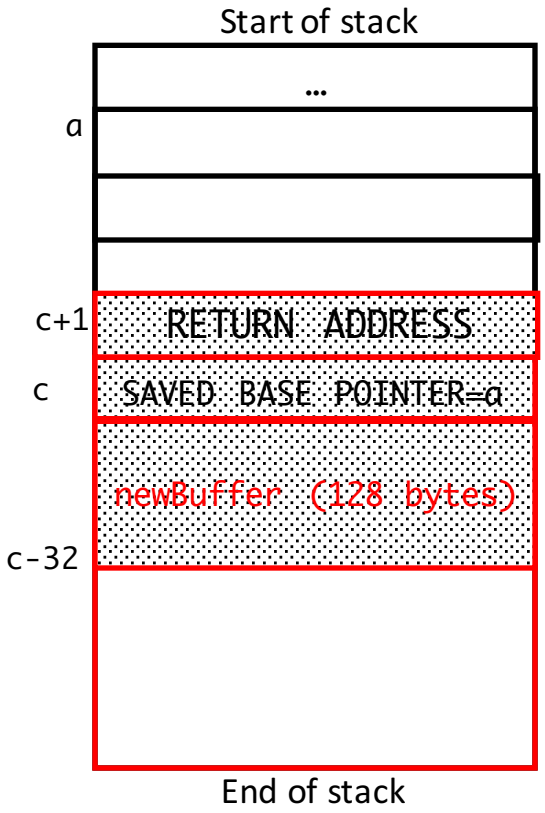


Imagine now that newfunc allocates a buffer of 64 bytes in memory

```
char newBuffer[128];
```

To newBuffer will be allocated 128 bytes of memory. In 32 bits architecture that corresponds to 32 memory cells (32 bits/cell=4 bytes/cell → 128/4=32)

Buffer overflow – attack (x86 32 bits) cntd



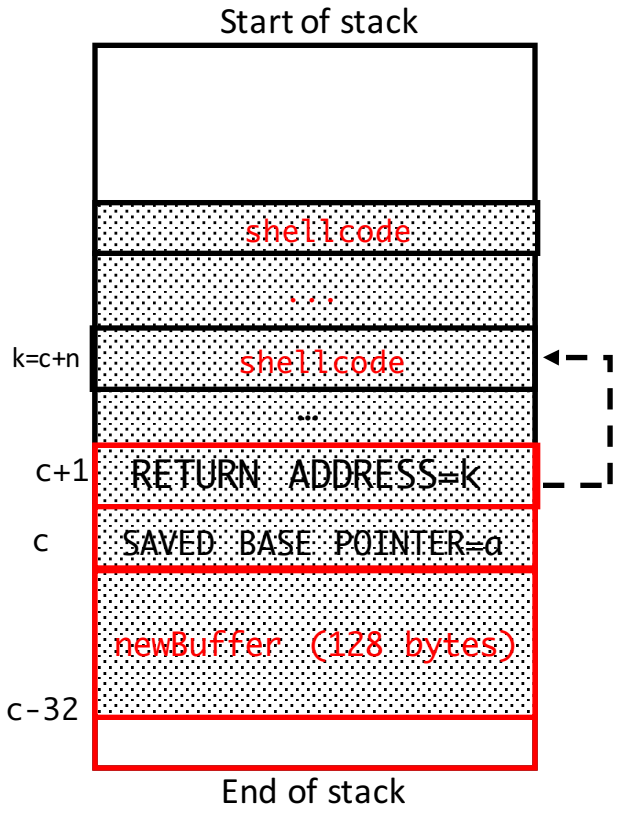
What happens if, without any control, newBuffer gets instead 128+8 bytes = 136 bytes?

newBuffer now overwrites

- SAVED BASE POINTER=a [addr(c-1)]
- RETURN ADDRESS [addr(c)]

This will typically throw a segmentation fault error as neither the saved base pointer nor the return address will likely contain valid values

Buffer overflow – attack (x86 32 bits) cntd



Let's take it a step further.

What happens if an attacker forges newBuffer in a more clever way?

Attacker can overwrite the return address in such a way that when the function returns the execution will jump to their own code.

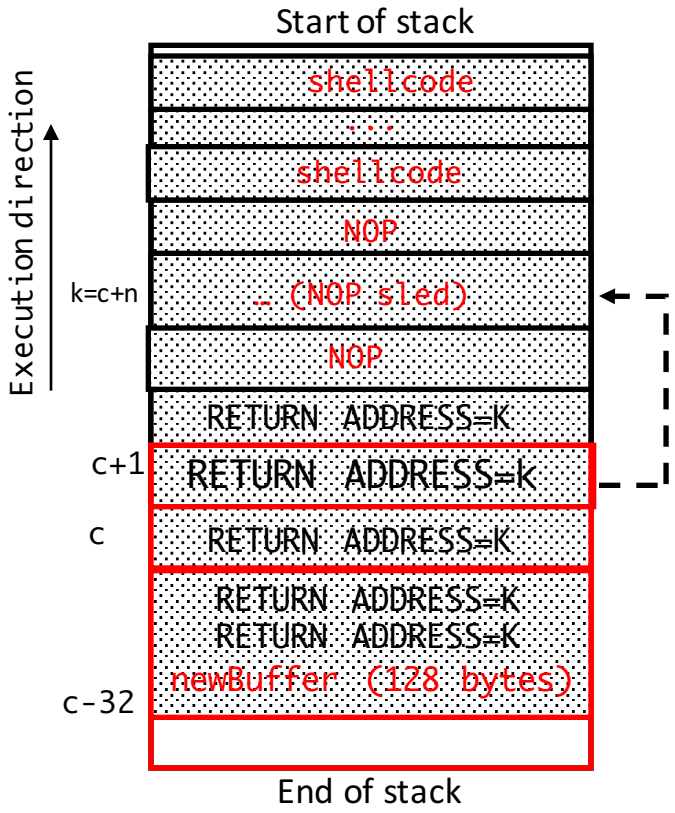
All the attacker has to do is to figure out the correct offset from the buffer to the location of the return address and the correct address for their own code

The attacker's code is commonly referred to as shellcode.

Once the address of the buffer is known it is trivial to find the address of the return address and set it correctly to point to the shellcode.

But memory allocation is not necessarily an entirely deterministic process.

Buffer overflow – attack (x86 32 bits) cntd



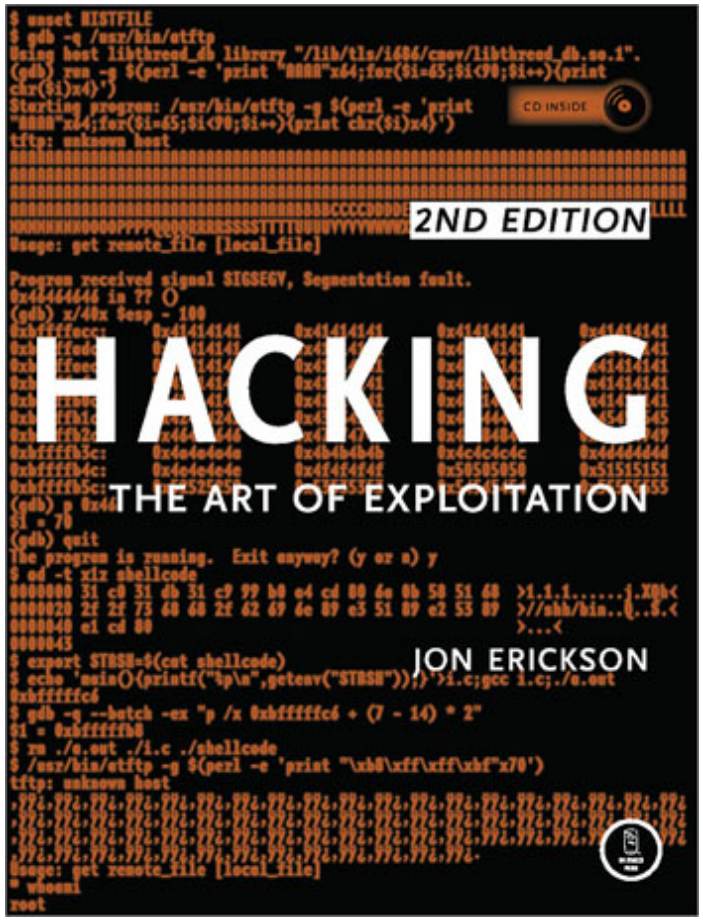
If attacker can not predict the return address exactly, then he does not know with precision

- where NewBuffer is relative to start of stack frame
- where the RET address is stored
- where the RET address should point at (i.e. where is the shellcode)

SOLUTION:
The attacker can employ a NOP (no-operation) sled on top of a sled of repeated RET addresses.

- Guesses that if he writes y bytes he will overwrite the RET
- Guesses in which range of memory addresses he can write, say c ± y
- He picks an address in that interval (e.g. k>c) and sets RET=k
- He forges the input in such a way that in the area around address k there are only NOPs
  - Instruction Pointer (IP) increases and nothing else happens
- On top of NOP sled he places his shellcode
  - As IP increases, the shellcode will eventually be executed

Reference to technical details



Buffer overflow → variants

- Return-to-libc
  - Instead of writing your code to execute, call a function that will do it for you
    - Re-use existing code
  - RET=addr(libc)
  - execution passes argument to libc from stack
    - e.g. "/bin/sh" → returns shell
- "Exec-before-return"
  - Instead of writing the RET (which pops the stack when context is switched) overwrite other parameters
    - E.g. EBP, other registers
    - Requires more in-depth analysis of assembly code
- Forge frame
  - You can forge a fake stack frame in the buffer
  - Modify EBP such that it will point somewhere in the buffer as if it was a stack frame (off-by-one buffer overflows)
  - Put your code in there

BoF – Causes

- There is no notion of “string length” in C
  - Strings are terminated by a ”null character” NUL → \0
  - No info on string length in memory
- Many default functions in C do not implement additional controls
  - strcpy(char *dest, char *src); gets(char *s)
  - Programmer needs to implement these on their own
- No distinction between executable and read-only sections of memory (x86)
  - Now mitigated in recent architectures