

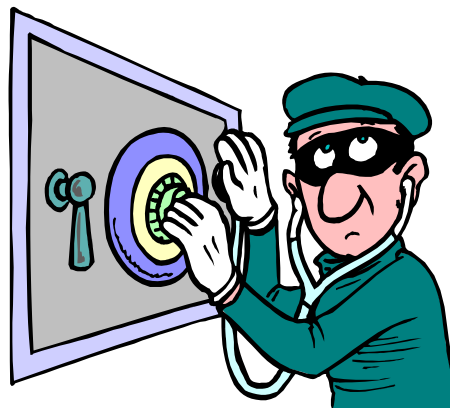
Exploits Against Software and How To Avoid Them

Nicolas T. Courtois



- University College London

Goals of Attackers



Break In

Stage 1:

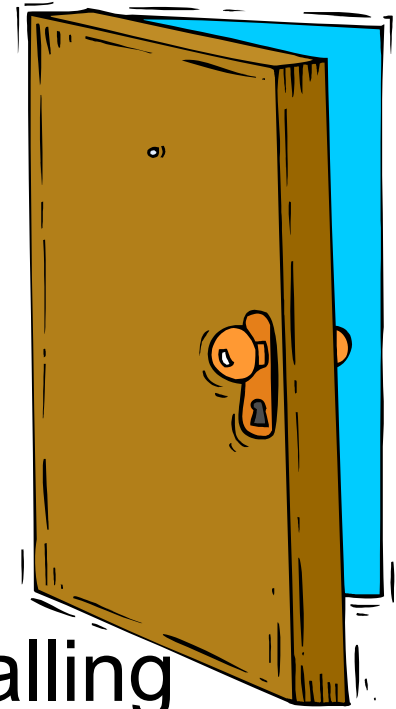
Get to run some code
(even without privileges).

Stage 2:

Gain admin/TCB access, usually by calling
other local programs and exploiting their
vulnerabilities.

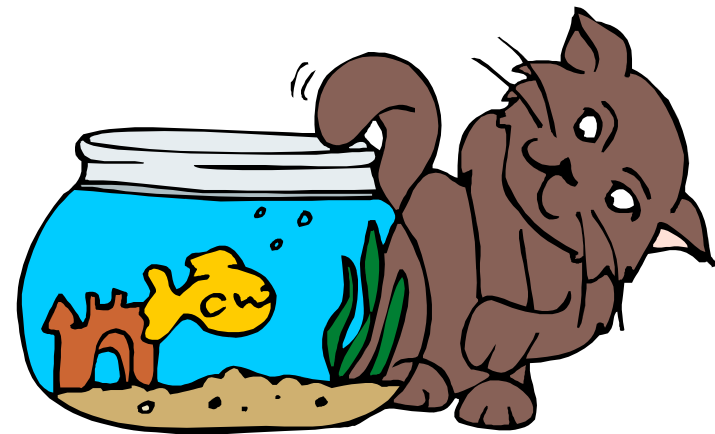
Stage 3:

Exfiltrate data, encrypt, and ask for a ransom
payment in bitcoins etc.



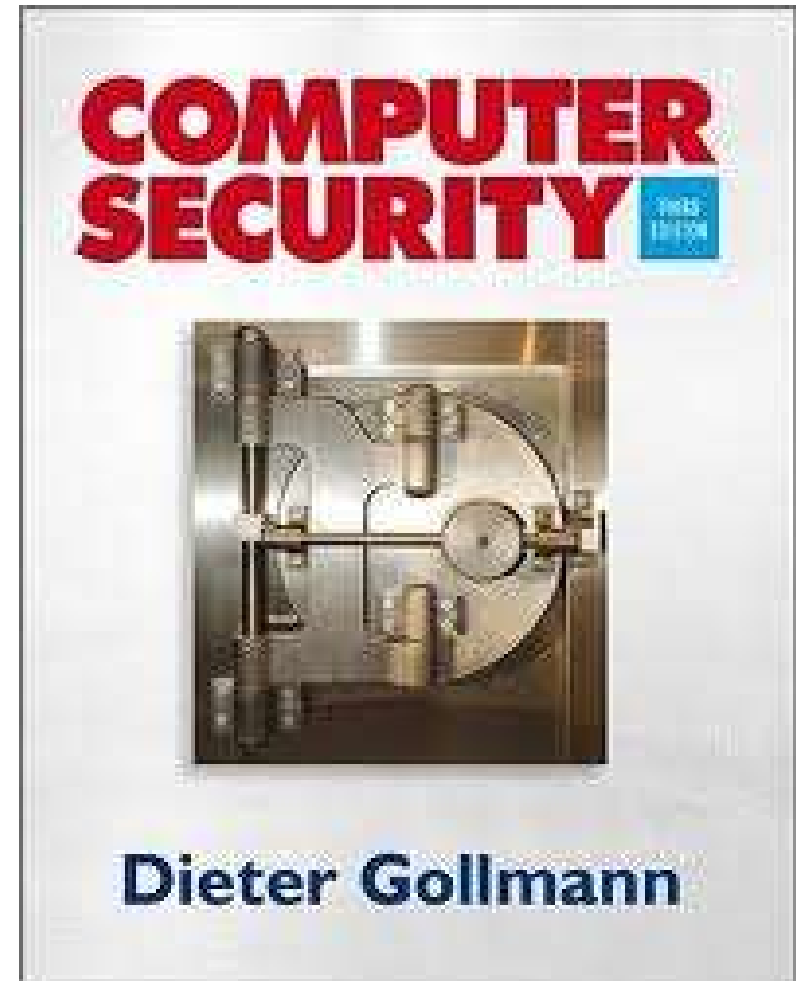
Goals for Software Exploits

- crash software (can be DOS)
 - crash/infect hardware
(e.g. hard drive, USB systems and devices)
 - get some data or side channels
 - **inject arbitrary code** (up to TCB access)
- these also happen accidentally...

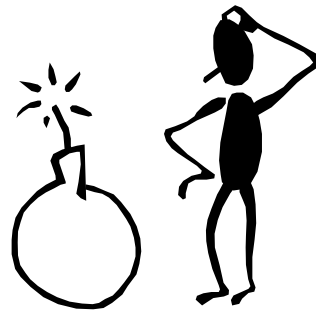


How to Break Into Computers?

- Stack attacks: Chapter 10.4.
- Defences: Chapter 10.7.
- Chapter 14: Software Security



What's Wrong?



Software Vulnerabilities

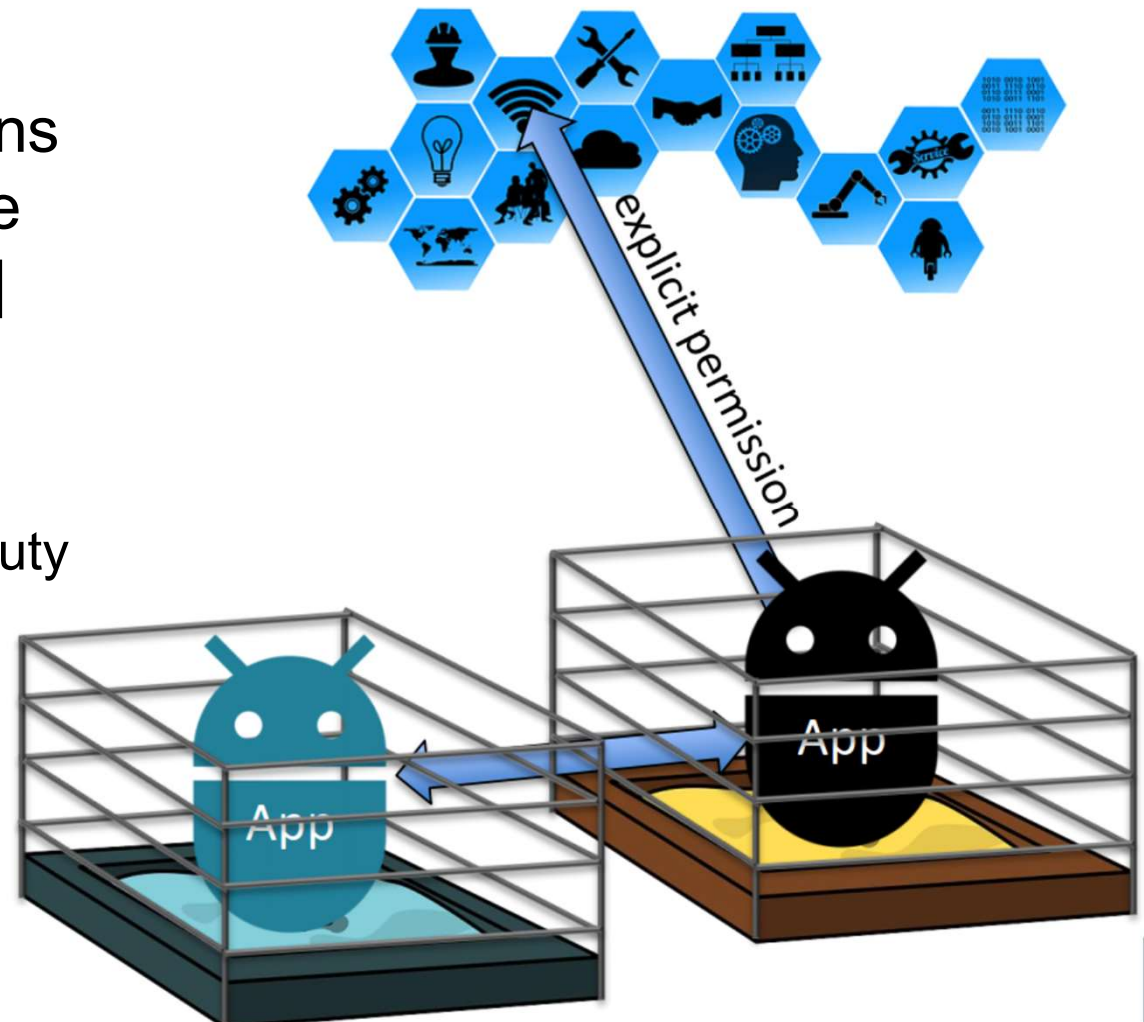
- Buffer overflow
- Input validation problems
- Format string vulnerabilities
- Integer overflows,
- CPU bugs
- Failing to handle errors / exceptions properly

Principles at stake:

- Usability/business contradicts security almost always
- Complexity, Common Mechanism,
- Same origin policy

*Android is NOT Like Linux

- Each application has a different set of explicit permissions
- Since 2015, permissions are awarded at runtime [previously install time]
- Better Least Privilege
 - no more Confused Deputy and Ambient Authority



Attack Families

- Microsoft “**STRIDE**” categorization of threats from design / implementation errors
 - Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege
- Common Weaknesses Enumeration (**CWE**)
a database of software errors leading to vulnerabilities.
See
CWE/SANS Top 25 Most Dangerous
Software Errors (retrieved 2019):
<http://cwe.mitre.org/top25/index.html>

CWE part 1

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	CWE-20	Improper Input Validation	43.61
[4]	CWE-200	Information Exposure	32.12
[5]	CWE-125	Out-of-bounds Read	26.53
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	CWE-416	Use After Free	17.94
[8]	CWE-190	Integer Overflow or Wraparound	17.35
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	15.54
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
[11]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
[12]	CWE-787	Out-of-bounds Write	11.08
[13]	CWE-287	Improper Authentication	10.78
[14]	CWE-476	NULL Pointer Dereference	9.74
[15]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.33
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type	5.50

CWE part 2

Rank	ID	Name	Score
[15]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.33
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type	5.50
[17]	CWE-611	Improper Restriction of XML External Entity Reference	5.48
[18]	CWE-94	Improper Control of Generation of Code ('Code Injection')	5.36
[19]	CWE-798	Use of Hard-coded Credentials	5.12
[20]	CWE-400	Uncontrolled Resource Consumption	5.04
[21]	CWE-772	Missing Release of Resource after Effective Lifetime	5.04
[22]	CWE-426	Untrusted Search Path	4.40
[23]	CWE-502	Deserialization of Untrusted Data	4.30
[24]	CWE-269	Improper Privilege Management	4.23
[25]	CWE-295	Improper Certificate Validation	4.06

Some Recent Bug Bounty Programs

Company	Model	Min pay	Max pay
Apple	Invite-only	-	\$200,000
Facebook	Open	\$500	-
Github	Open	\$200	\$10,000
Google	Open	\$300	\$31,337
Intel	Application	\$500	\$30,000
Microsoft	Open	\$500	-
US Pentagon	Pilot run	\$100	\$15,000
Tor Project	Open	\$100	\$4,000
Uber	Open	-	\$10,000
Wordpress	Open	\$150	-

List from tripwire.com (Commercial Integrity Software)
<https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/10-essential-bug-bounty-programs-2017>)

Vectors of Attack - Inputs

Software Input Exploits -Targets

Exe programs:

- command line arguments
- environment variables
- configuration files / settings changed in the registry by another program...
- network packets
- RPC or API or shared memory [e.g. bitcoin client].

Windows dlls / Unix runtime precompiled libraries:

- function calls from other programs

Exploit against Linksys router [2004]

It has a ping utility. Pretty innocent?

The attacker types:

```
127.0.0.1 | ls> /tmp//ping.log
```

Danger of Environment Variables

In UNIX:

- Set `LD_LIBRARY_PATH` system variable to avoid the standard precompiled libraries...
- Hacker puts his own libraries in his own directory...

Fix: modern C runtime libraries in Unix stopped using `LD_LIBRARY_PATH` variable when the `euid` is not the same as the `ruid`...(like `pwd` program).
Normal case, 99.999% of the time.

****Recall:

In Unix each process has several user IDs:

- Real User ID == ruid, identifies the owner of the process
- Effective User ID == euid, determines current access rights

set-uid programs

Definition:

A “set-uid program” (property acquired at creation/compilation/installation time) is a program that assumes the identity and has privileges of the owner of the program, though a different user uses it.

Examples:

- passwd
- su, sudo

BTW: if copied to a “user” directory, they will stop working!

Injection Attacks vs. Path

- path traversal or path abuse...
- what if my path is or contains “.”?
 - See later slides.
- what if the user can write files of their choice and uses “../../filename”?
 - Quiz: what if a Microsoft compiler is given this path???? any special privileges?
How do we call this sort of attack? C.D.

*Path Traversal

```
#!/python
import glob
import os.path

import cherrypy ## Need cherrypy web framework
from cherrypy.lib.static import serve_file

class Root:
    def index(self, directory="."):
        html = """<html><body><h2>Here are the
files in the selected directory:</h2>
<a href="index?directory=%s">Up</a><br />
""" % os.path.dirname(os.path.abspath(directory))

        for filename in glob.glob(directory + '/*'):
            absPath = os.path.abspath(filename)
            if os.path.isdir(absPath):
                html += '<a href="/index?directory=' + absPath + '>' \
                    + os.path.basename(filename) + "</a> <br />"
            else:
                html += '<a href="/download/?filepath=' + absPath + '>' \
                    + os.path.basename(filename) + "</a> <br />"

        html += """</body></html>"""
        return html
    index.exposed = True

class Download:
    def index(self, filepath):
        return serve_file(filepath, "application/x-download", "attachment")
    index.exposed = True

if __name__ == '__main__':
    root = Root()
    root.download = Download()
    cherrypy.quickstart(root)
```

CWE-22

Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Exercise:

Cherrypy web framework documentation, on how to implement file downloads

- (1) What is going on here?
- (2) Find the bug.
- (3) Why is this a case of a confused deputy?
- (4) How do you fix it?

Code Injection Attacks

- Code injection (e.g.: the `eval` function)
- OS Command Injection
 - In C/C++ we use
`system(some Unix or Windows command in a char * buffer);`
 - `os.system("procmail %s" % user_input)`

More Attacks on PATH in Unix

Now imagine that any “setuid program” contains the following line:

```
system("ls ... ");
```

OOPS...

there are several ways to use this to run any program as root...

More Attacks on PATH in Unix

A “setuid program” ABC contains the following line:

```
system("ls ...");
```

The user sets his PATH to be “.” and places his own program **ls** in this directory.

This program will be run as root!

Can this be done remotely?

- In PHP language, used by all web servers, they have `PASSTHRU()` function that executes arbitrary code...
- Assume it contains a user input that comes from the web client browser.
- insert “`; command231`” or “`| command231`”.
- This will make the server execute `command231` and output the result to the web page displayed.

PHP have later banned this
and many other things from the PHP language...

Another Classical Exploit in Unix

- the IFS variable: the characters that the system considers as white space
- - now add “s” to the IFS set of characters
 - `system(ls)` becomes `system(l)`
 - a function `l` in the current directory will be run as root...

Same Origin Policies

- Applies to scripts that run in browsers
- Applies to browser tabs/windows.
- When servers are manipulating cookies.
- Origin = domain name + protocol + port
 - all three must be equal
 - however,
some access may be allowed for pages
from same domain, but not same host

Threats:

- Impersonation of a Legitimate User, Session Hijacking
 - violating the trust a website places in a remote user, allowing the attacker to initiate HTTP requests in the context of the remote user or impersonate the remote user entirely [e.g. continue connection to a bank]
- Impersonation of a Legitimate Website (Phishing)
 - violating the trust a user places in a remote site by impersonating the site in whole or in part
 - e.g. subtle MIM attacks, typically . the user thinks A, the server view is B

Attacks Against Web Servers

- (SQL) injection attacks: confusion data/code.
- Cross site request forgery (CSRF) == Session Riding
- Cross site scripting (XSS)

Unix and Web Servers

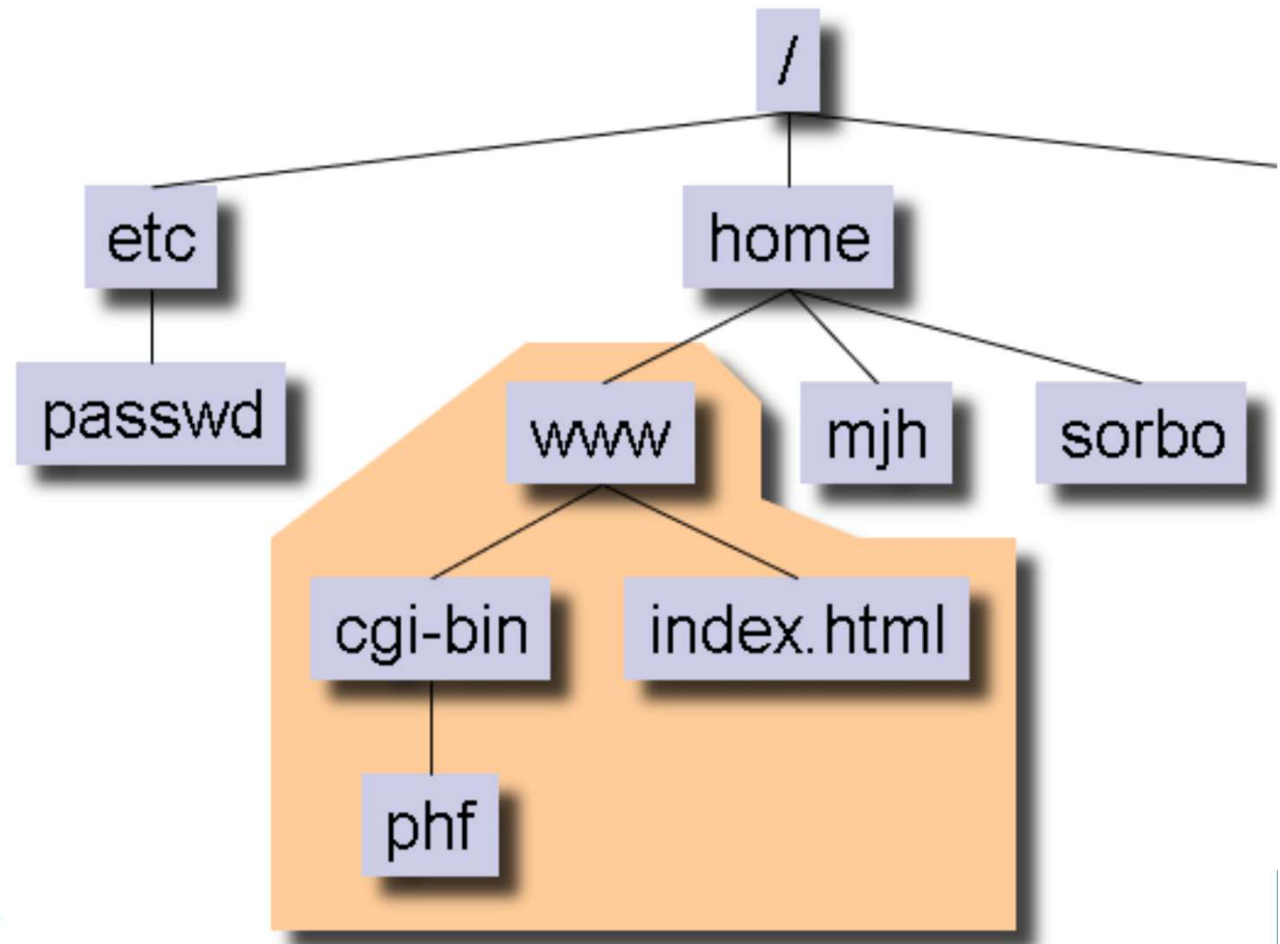
- Old times: running as root. Needed to open port $80 < 1024$.
- Exploit the web server => you become root.
- Solution: **containment \neq prevention**:
limiting the powers of the server.

Dropping Privileges

- Unix provides several ways to drop privileges:
 - **setuid(nobody)**
 - **chroot()**
 - FreeBSD's **jail()**
 - OpenBSD's **sysrtrace()**

chroot(/home/www)

- change the effective root directory:



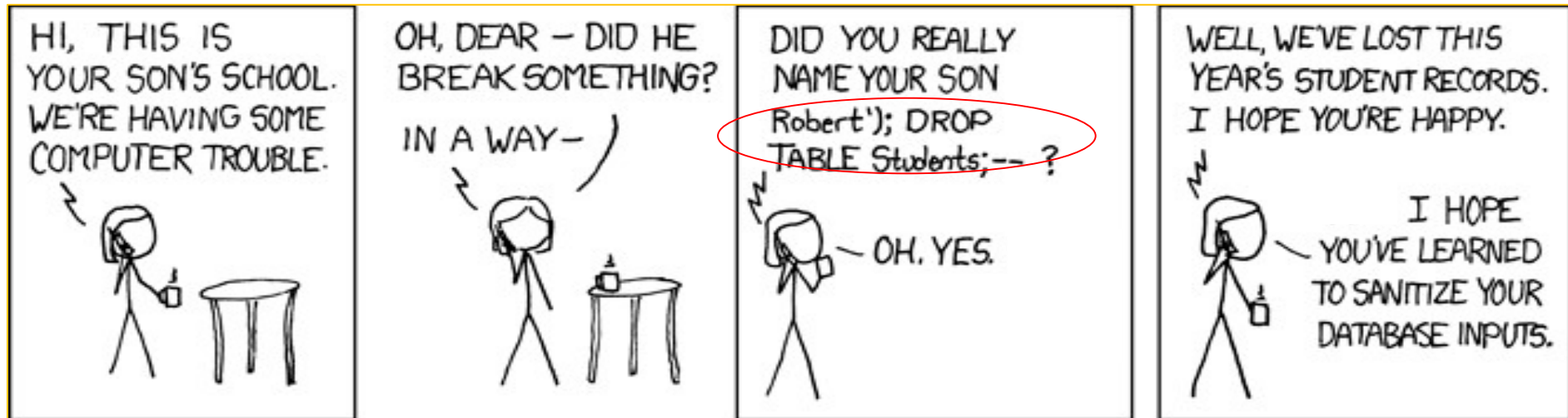
jail()

- **chroot()**
- FreeBSD's **jail()**: same but stronger:
 - No IPC outside of jailed apps
 - Even root cannot load Kernel modules, create device files or affect “real” machine e.g. cannot reboot.

systrace

- OpenBSD's `systrace()`:
limits which system calls may be used.

SQL Attacks!



<https://xkcd.com/327/>

- (SQL) injection attacks: **confusion data/code**.

[1] CWE-89: Improper Neutralization of Special Elements used in an SQL Command

== 'SQL Injection' ==

- “Insecure Interaction Between Components”
 - Application ↔ SQL database.
 - The bug: the application accepts some low integrity input. It then constructs an SQL query based on that input without sanitizing it.
 - Result: the SQL database interprets it, in a way unintended by the application.

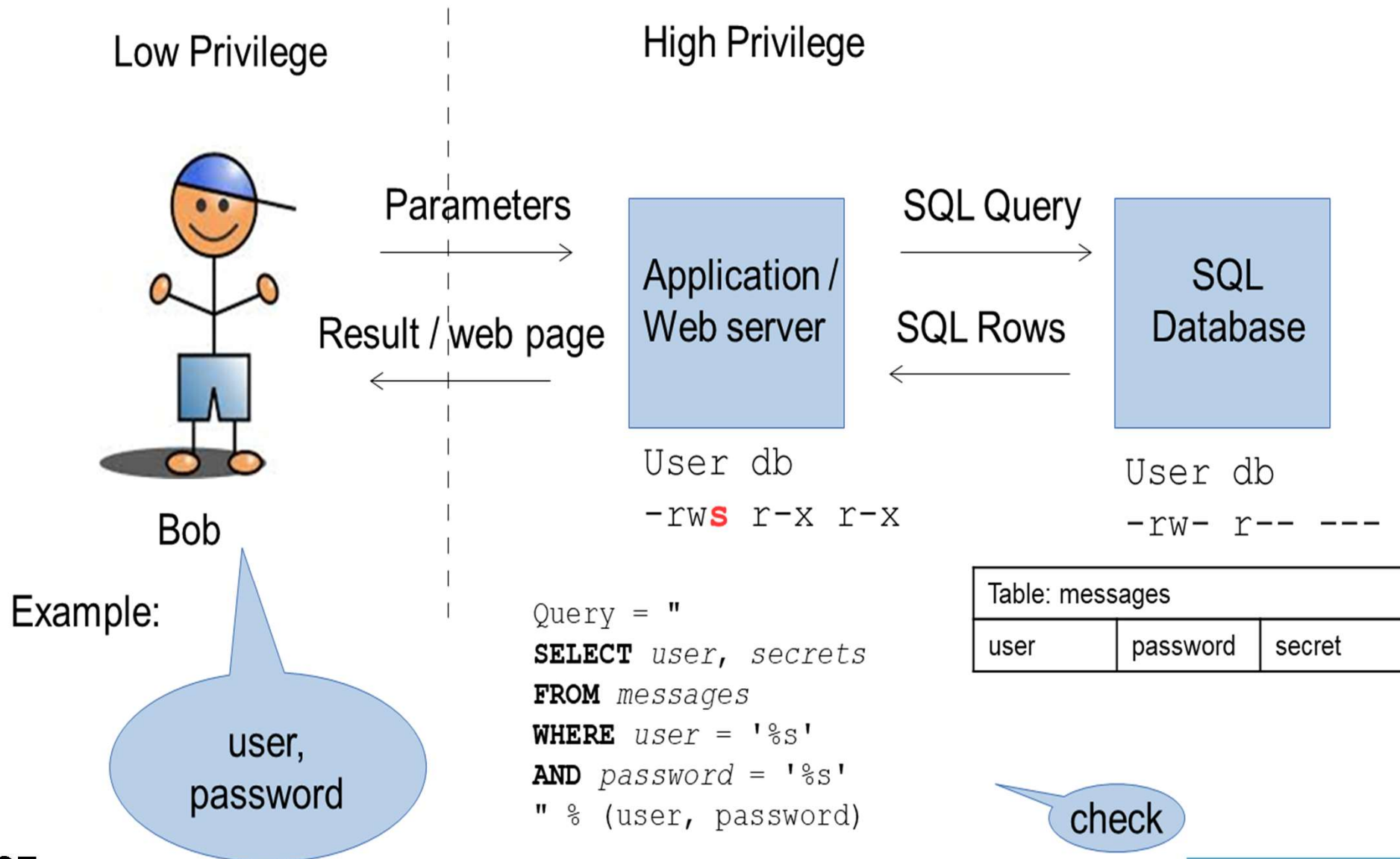
***Related to:

- [2] CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- [4] CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- [9] CWE-434 Unrestricted Upload of File with Dangerous Type
- [12] CWE-352 Cross-Site Request Forgery (CSRF)
- [22] CWE-601 URL Redirection to Untrusted Site ('Open Redirect')

SQL Injection - Attacker Goals

- Specific attacks:
 - extract secret information (confidentiality)
 - Corrupt/delete other user's records (integrity)
- Generic attacks (Elevation of Privilege):
 - Read the full password table (→ log in as admin).
 - Insert a new admin account with known password.
 - Modify the permission table to make yourself admin.

SQL Injection Setting



Motorway Toll Pay As You Go?

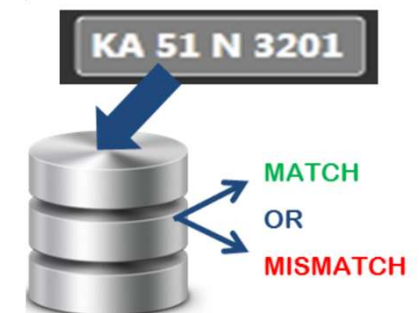


PHOTO: JOHN R. COUGHLIN/CONNMONEY

?



- Imagine a toll station with a camera with OCR and direct database access:
 - Car with number plate 'OR 1=1; --
 - Car with number plate '; DROP TABLE cars; --



***Classical SQL Injection

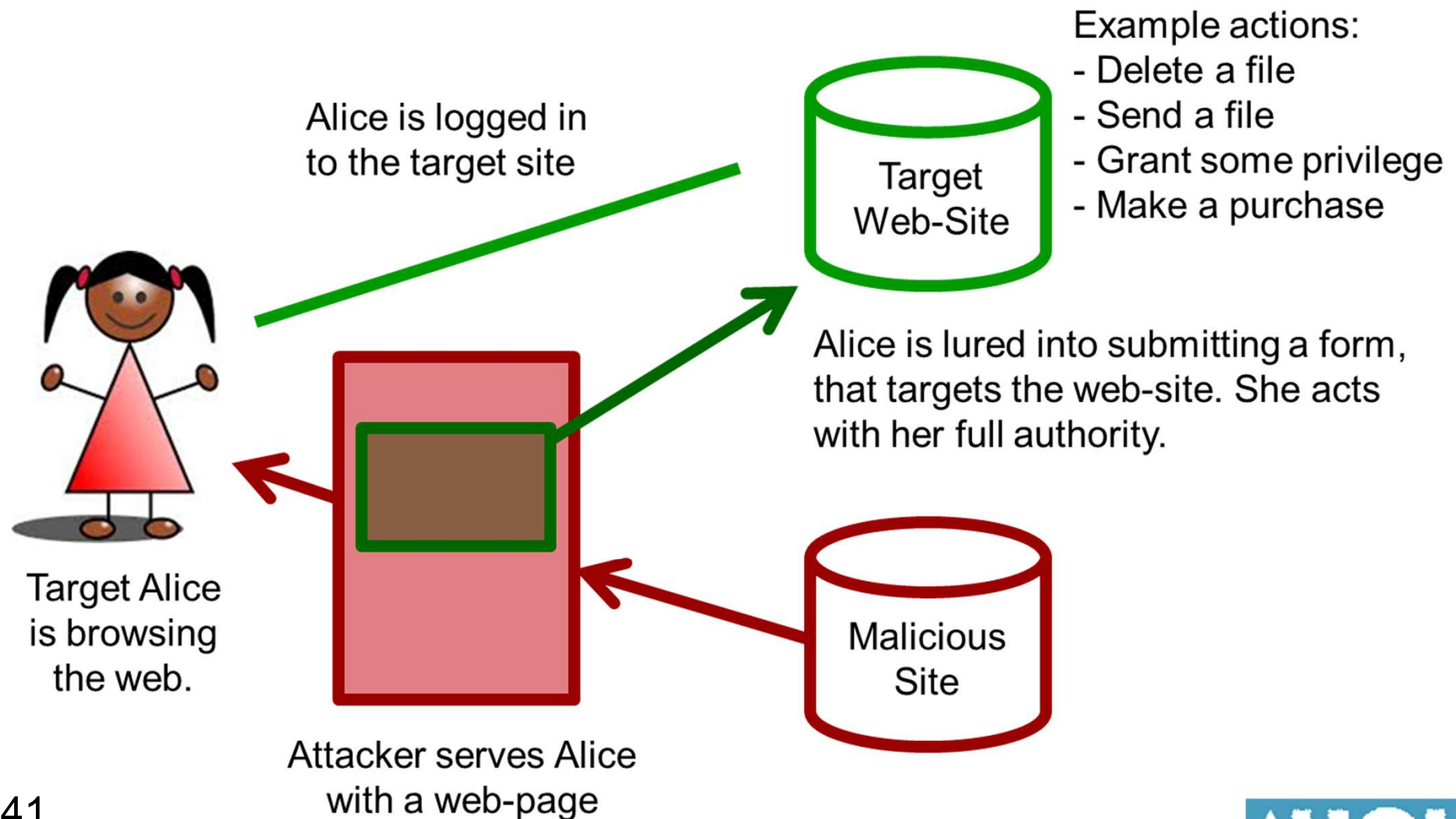
```
function check_user($user, $password) {
    if (connect()) {
        $password = substr($password, 0, 8);
        $sql = "select * from users where us = '$user' and pw = '$password'";
        $result = mssql_query($sql);
        if (mssql_num_rows($result) == 1) {
            setcookie("user",$user); setcookie("password",$password); return 1;
        } else {
            ?> <h3>Sorry, you are not authorized!</h3> <? return 0;
        }
    }
}
```

- What is the vulnerability? Provide \$user and \$password!
 - Log in with \$user="admin", \$password="' OR '1'='1'" or even \$password="'; DROP TABLE users; "
- What now?
 - Use \$user="'; DROP TABLE users; --"
 - Or: \$password="'OR' '='"

CWE-352: Cross-Site Request Forgery (CSRF)

- “Insecure Interaction Between Components”
 - Web-client ↔ Web-Server
 - The bug: a web-client is confused by an adversary into submitting a request to an honest web-server using the client credentials with the web-server.
- When is that an issue?
 - Web-server relies on user's identity for some actions.
 - Web-server accepts actions in forms or URLs.
 - Attacker can determine the right inputs for all forms.
 - Attacker can lure the victim to a malicious page while logged on.

Cross-Site Request Forgery Setting



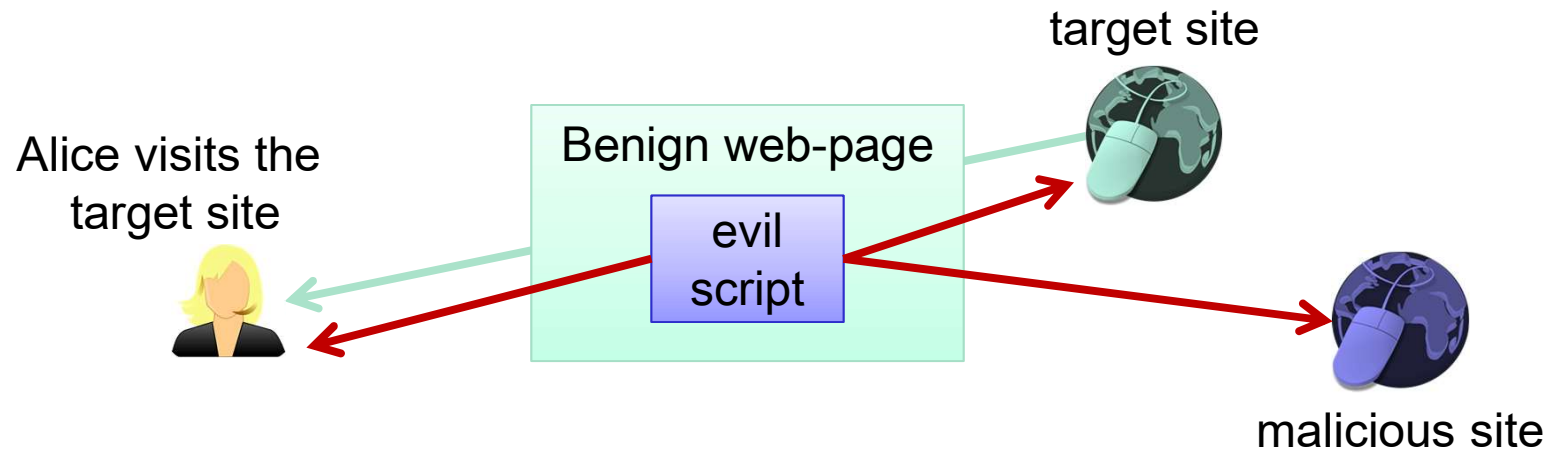
Cross-Site Request Forgery Principles

- Confused Deputy
 - Alice's web-client is confused into performing an action that seems to be authorized by Alice, but that in fact grants privileges held by Alice to the adversary.
- Ambient Authority
 - The web-client security model and authentication based on “cookies” always acts with the privileges of Alice when interacting with the web-server she is logged in.

*Mitigations

- Options: confirm origin of authority and request.
 - Make “GET” requests side-effect free.
 - Include within each (valid) form an authenticator that the adversary cannot guess. Check for the authenticator before acting on a request.
 - Check the HTTP “referrer” or “origin” field of the request before executing it.
 - Request re-authentication for every action.
- Why is all this so hard?
 - HTTP requires web developers to re-define a session layer for each application. No standard way of managing sessions → errors.

Direct Cross Site Scripting (XSS)



- Alice visits a benign webpage. Attackers add a malicious script to this site, e.g., disguised as useful component, or an advertisement (sometimes called “**Malvertising**”).
- Alice’s browser trusts the script (it is from the benign site).
- Consequently, the script runs under the privileges of the benign site.
 - Problem: The script can access the cookie, e.g., send this cookie to the adversary.
- How can you get the script into the site?

Cross Site Scripting in Practice: Reflective XSS



- Lure Alice into sending a script to the server (e.g., as her “name”).
- The server reflects back the script to Alice (e.g., shows the “name”).
- Alice’s browser gets the webpage with the script and executes it.
- Combination: 2 components: malicious link to click and “innocent” Javascript.
- `http://www.example.com/welcome.html?name=Joe`
- `http://www.example.com/welcome.html?name=<script>alert(document.cookie)</script>`

Buffer Overflow since 1972!!!

Software Buffer Overflow Exploits

I will explain in details only 1 type of buffer overflow attack...

Stack Smashing

There are many other types of software vulnerabilities...

Study of these requires a lot of technical expertise about programming, compilers, assembly and CPUs...

Buffer Overflow History

Extremely common since the 1980s.

Consistently about 50 % of all CERT advisories.

Usually leads to a total compromise of the machine...

****Example: CWE-2011-1938**

```
PHP_FUNCTION(socket_connect)
{
    zval          *arg1;
    php_socket    *php_sock;
    struct sockaddr_in sin;
    #if HAVE_IPV6
    struct sockaddr_in6 sin6;
    #endif
    struct sockaddr_un s_un; /* stack var */
    char          *addr;
    int            retval, addr_len;
    long          port = 0;
    int            argc = ZEND_NUM_ARGS();
    [...]

    case AF_UNIX:
        memset(&s_un, 0, sizeof(struct sockaddr_un));
        s_un.sun_family = AF_UNIX;
        memcpy(&s_un.sun_path, addr, addr_len); /* Unlimited copy. Stack overflow */
        retval = connect(php_sock->bsd_socket, (struct sockaddr *) &s_un,
                        (socklen_t) XtOffsetOf(struct sockaddr_un, sun_path) + addr_len);
        break;
    [...]
}
```

Can Programmers Get It Right?

Lot of evidence around that **they cannot**.

- the behavior of Turing machines is very HARD to analyse,
 - cf. Rice thm.
- it is usually easier to rewrite code from the scratch than to find all bugs in it
- software economics, time to market, code re-use etc...

Major problems also occur at the compiler and runtime level...
(even CPUs have bugs that can be used for exploits).

Problems with C and C++

- C and C++ particularly dangerous
 - Fast, therefore used in servers and all critical code (performance-wise and security-wise)
 - allows arbitrary manipulation of pointers
 - but not outside the virtual 2 Gbyte space allocated by the OS

Software Under Attack

Main goal:

inject arbitrary code through standard input channels of the program.

Input-dependent vulnerabilities.

Excessively common in software we use every day... Unix and Windows alike...

Exploit =

specially crafted input

that allows a certain task to be accomplished
compromising the security policy
usually executing arbitrary code.

Goal: execute with the privilege level of the program:

- web server running as superuser...
- Ordinary programs running as user...

Furthermore, injected code may use another vulnerability to permit privilege escalation.

Buffer Overflow Attack: Stack Smashing and ASM Code Injection in C



Buffer Overflow in C

`char command[256]="";`
allocated from the stack.



Now imagine we input longer data than 256 bytes and use `strcpy(command,*input_data)`.

In theory: “undefined behaviour”..

In practice: we can predict what will happen.

historical roots

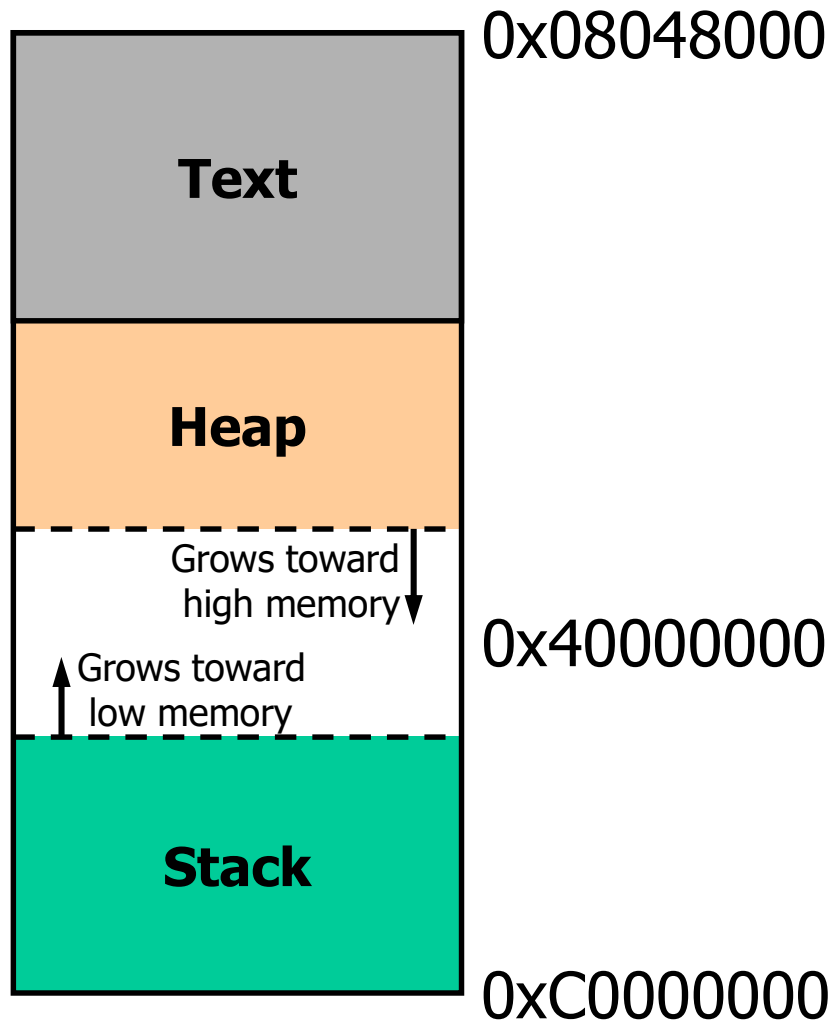
Since ever, in CPU assembly and in compiling structured programs, the habit is to save the state of the CPU when calling a sub-routine.

And saving the return address.

It is essential which comes first...
otherwise there would be no such attack.

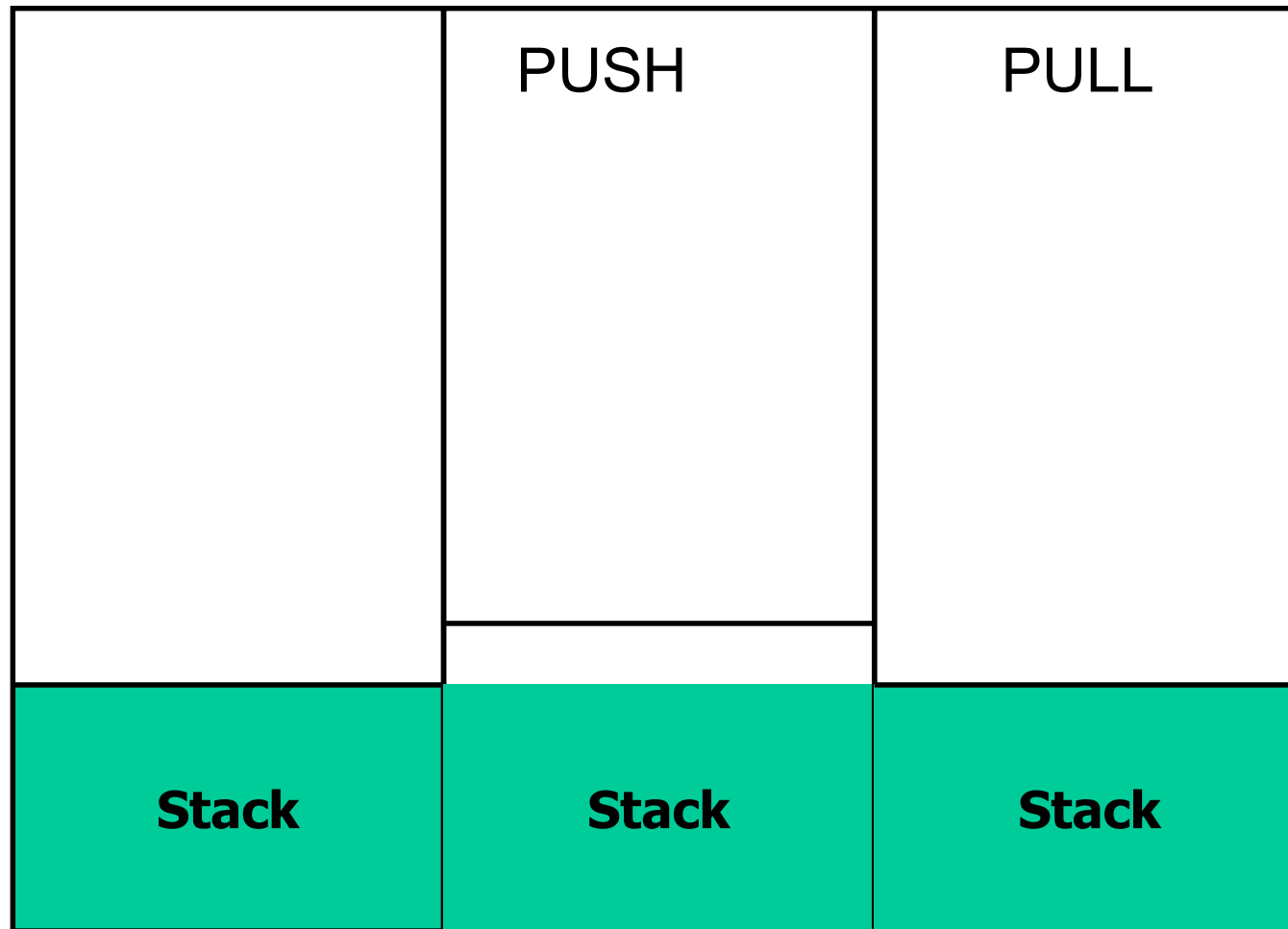
This is saved on the process **stack**.

Process Memory Layout



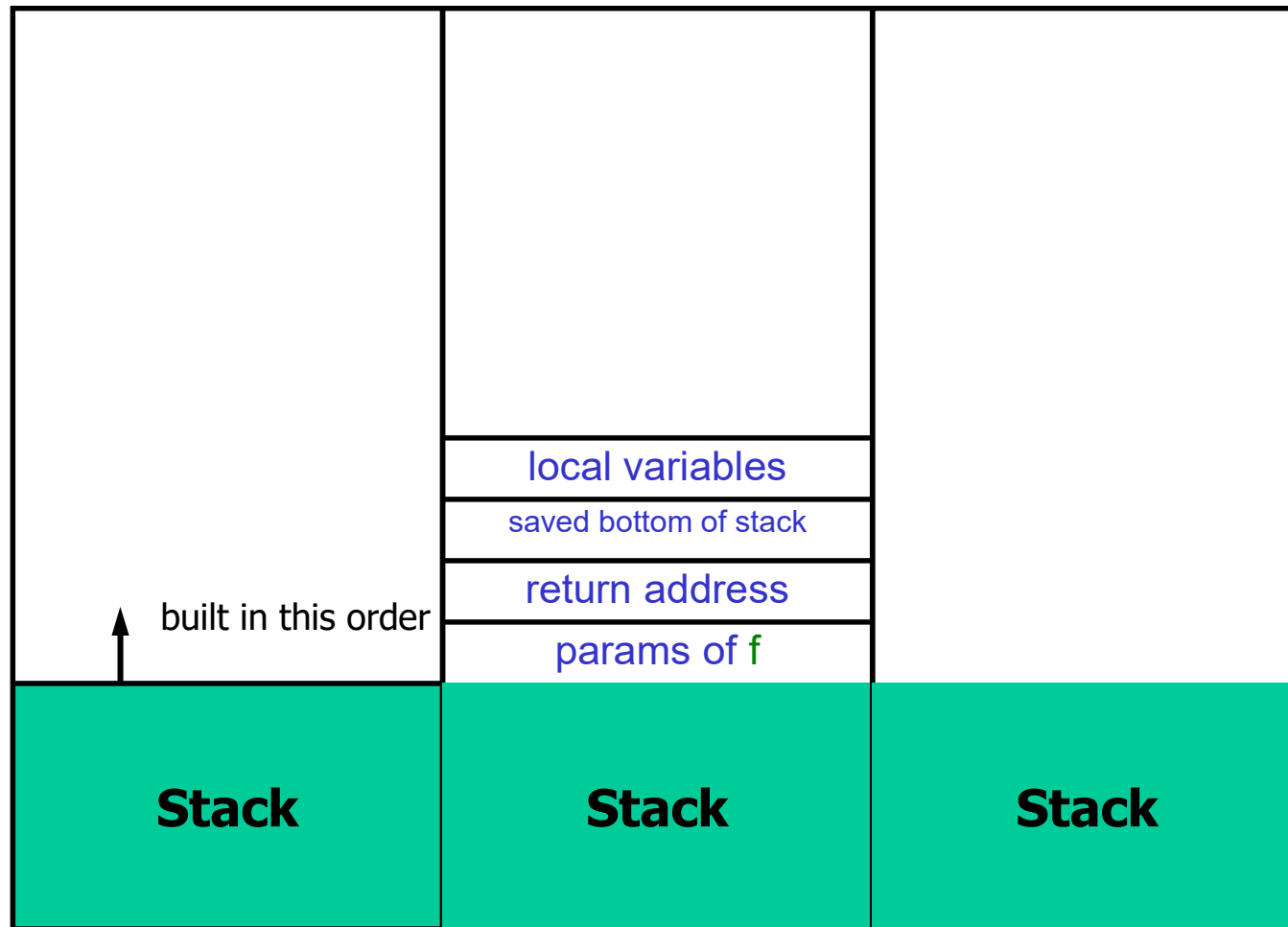
- Text: loaded from exec code and read-only data
- Heap: runtime allocated objects,
- Stack: LIFO, holds function arguments and local variables,

Calling a Sub-Routine in C



on every
CPU
since
ever...

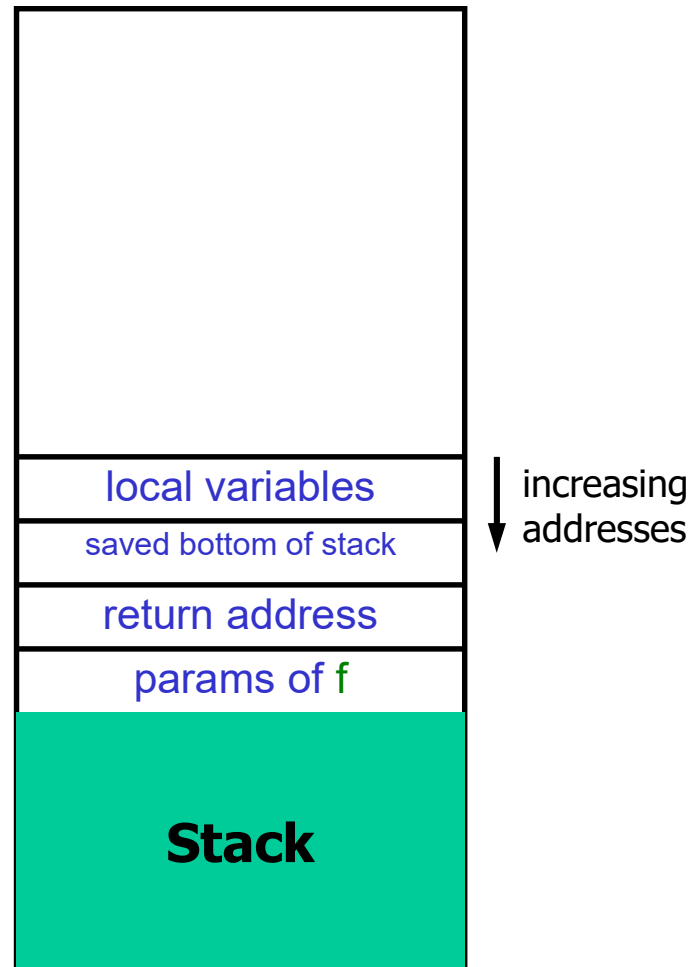
Stack Frames for one C Function *f*



exploit on f



overwrite



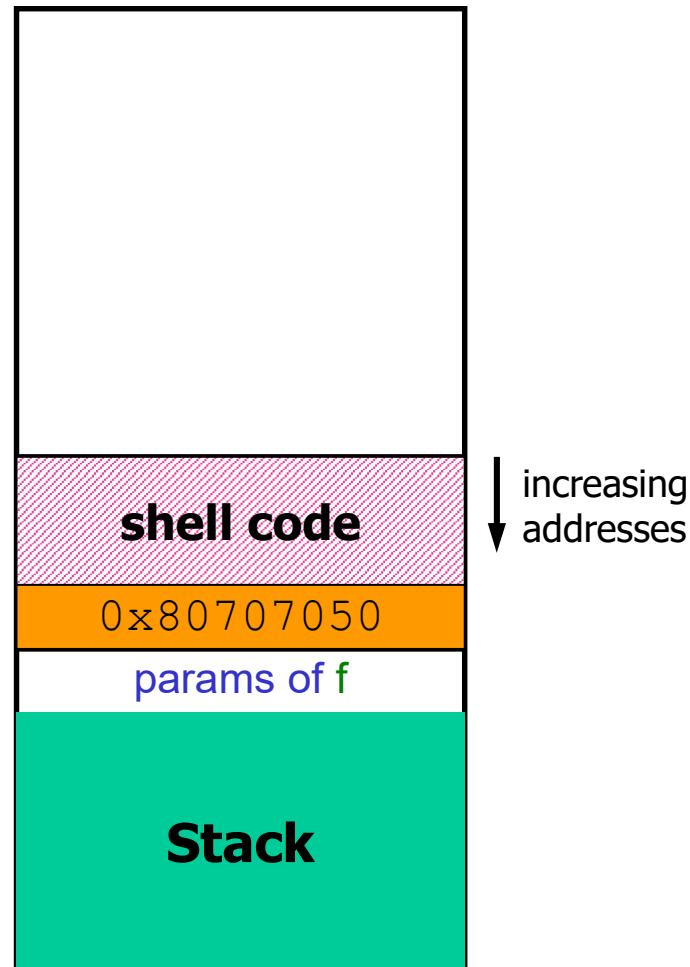
```
void f(params)
{
  char command[256]="";
  ...
  strcpy(command,sth)
}
```

size
easy to
guess

exploit on f



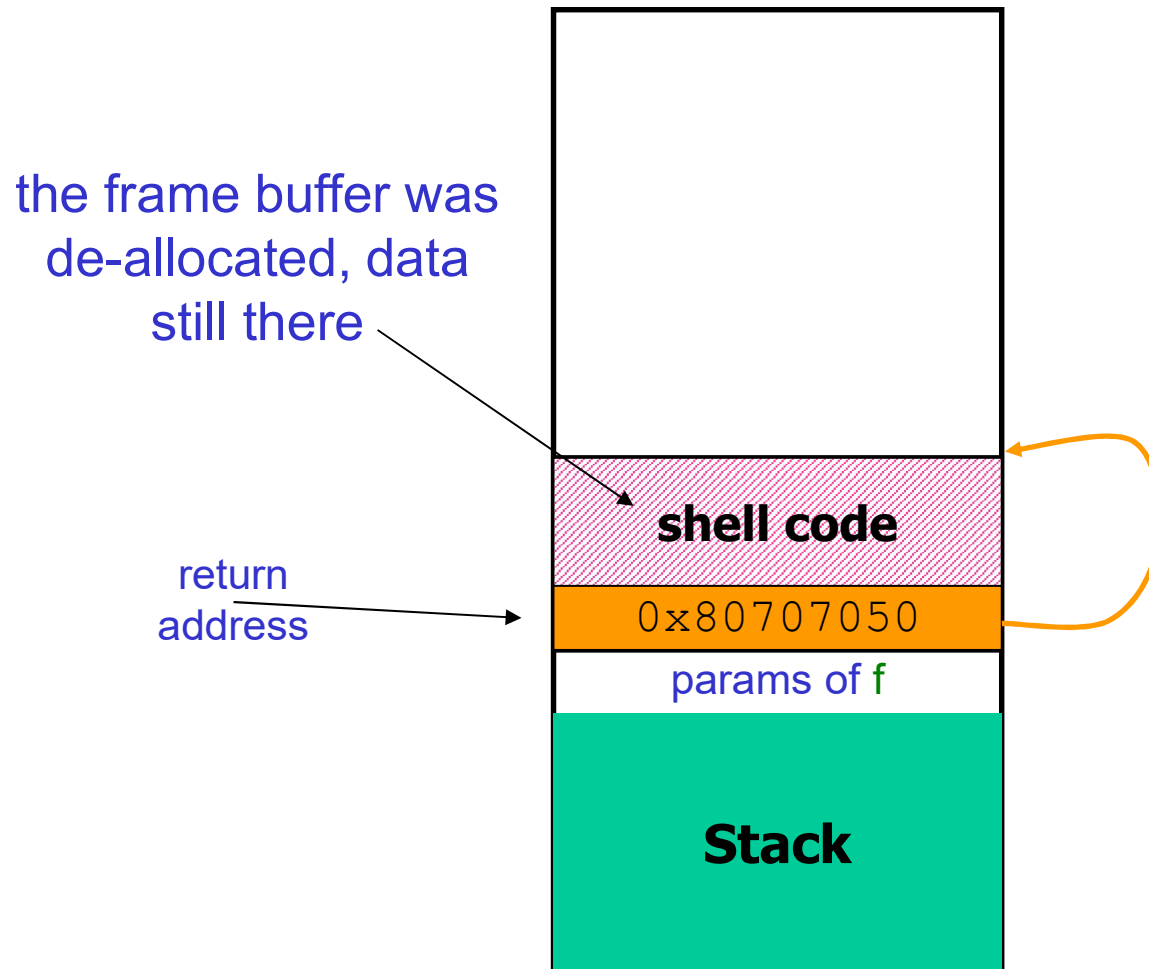
overwrite



```
void f(params)
{
  char command[256]="";
  ...
  strcpy(command,sth)
}
```

easy to
guess

when **f** finishes



Reliability

up to very high,
up to 100%

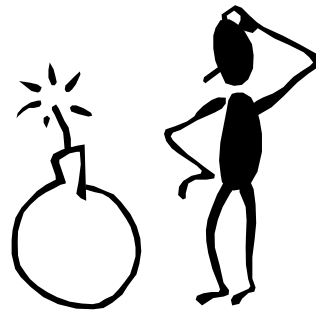
(there are stable exploits,
never ever fail
and produce consistent results)

Examples of code:

<http://shell-storm.org/shellcode/>

Hackers vs. Defenders

Advanced Exploits



Solutions (1)

- use type and memory safe languages (Java, ML)
- **clean** the de-allocated frame buffer: slow!!!

Partial solutions (not perfect)

- certain forms of access control?
 - yes, replace pointers by use of “un-forgable reference” tokens
- sandboxing and “secure” VM techniques.
- store things in a different order:

ASLR = Address Space Layout Randomisation – at the runtime!

- suddenly it makes a lot of sense to recompile the Apache web server software on each server. Reason: 75 K copies, Slammer worm.
 - OpenBSD (enabled by default)
 - Linux – weak form of ASLR by default since kernel 2.6.12. (much better with the Exec Shield patch for Linux).
 - Windows Vista and Windows Server 2008:
 - ASLR enabled by default, although only for those executables and dynamic link libraries specifically linked to be ASLR-enabled. So only very few programs such as Internet Explorer 8 enable these protections...

Solutions (2)

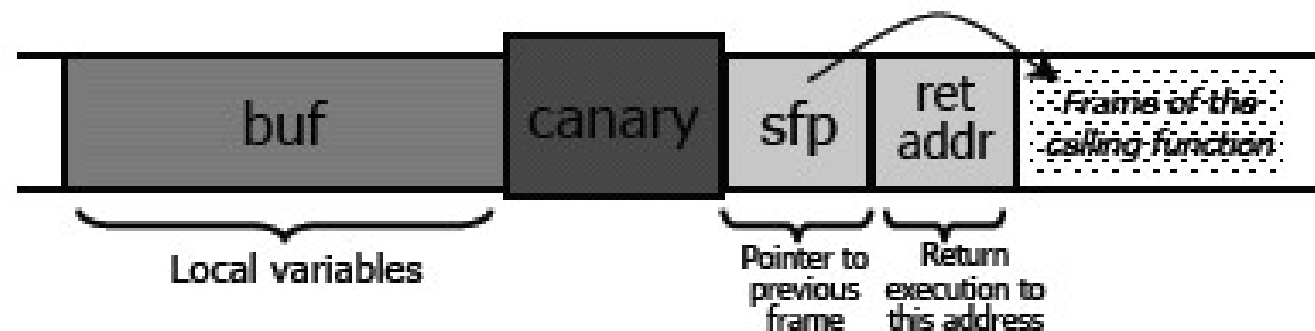
Automated protections with **canaries**:
store known data at the end of the buffer. Check.

- StackGuard, ProPolice, PointGuard
= extensions of GCC, automatic.
- similar protections also by default in MsVisual Studio.

Time performance overhead: about +10%.

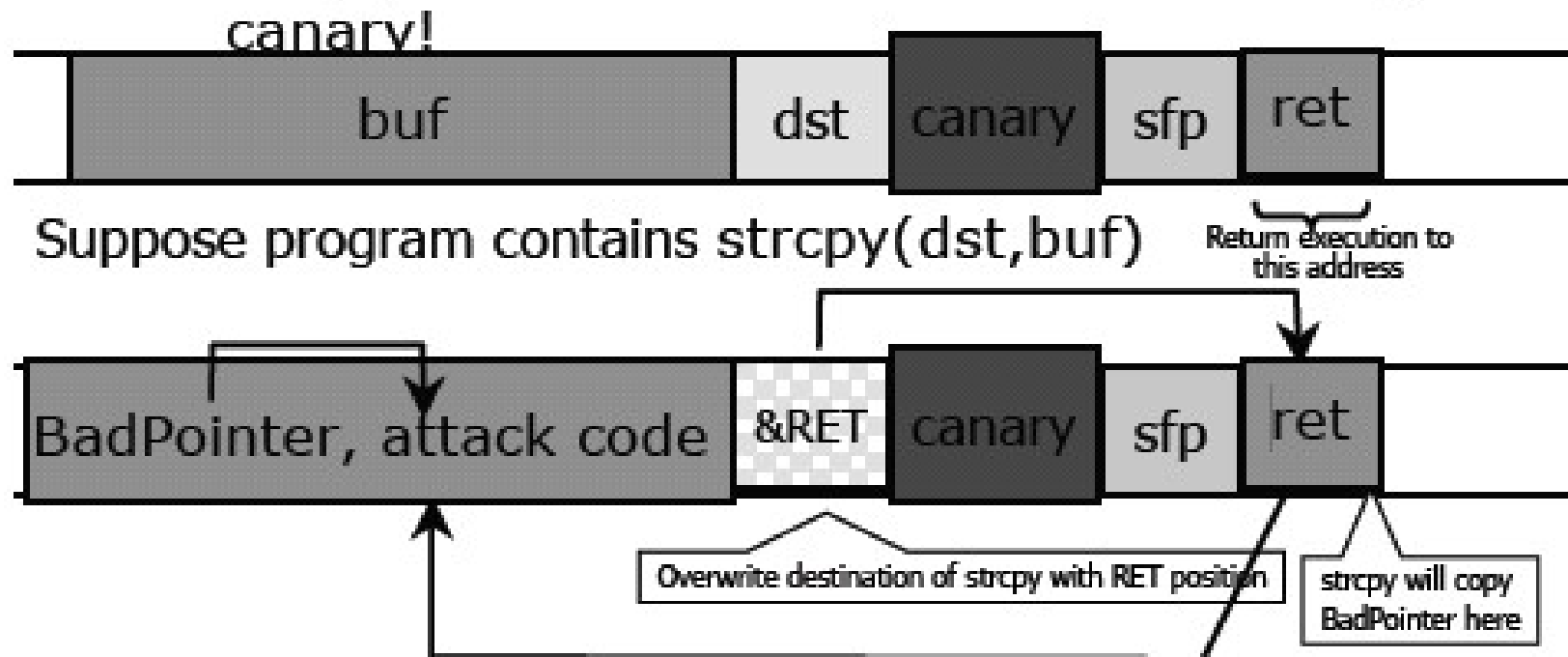
Is this secure?

- what value should the canary have?
 - what if the same C routine is called twice?



Attack Against StackGuard (Canaries)

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
 - strcpy will write into RET without touching



Solutions (3)

- hire a programmer with extensive understanding of software attacks
 - less attacks, will not eliminate them

Cheaper solutions:

- make sure that stack space is marked as impossible to execute ()
 - DEP = Data Execution Protection.
 - Linux PaX (a patch for Linux),
 - Supported in Windows XP SP2 too, not widely used yet.
 - Requires DEP, requires PAE mode.
- blacklist parts of C language!
 - ongoing process.

W \oplus X Page Protections - Unix

- CPU has page protection implemented in combination of hardware / OS kernel
 - for each 4K memory page, permission bits specified in page table entry in kernel: read, write
- **Exclusive OR**
 - Each page should be either writable or executable, but **not both**:
W \oplus X
 - exe program (a.k.a. text) pages: X, not W
 - data (stack, heap) pages: W, not X

Remark: In Linux PaX, for older processors, the mechanism of W \oplus X is implemented in a tricky way based on segment limit registers => memoryx2, negligible performance degradation.

DEP = Data Execution Prevention - Windows

The “X” idea:

Memory pages **MUST** be explicitly marked as executable to be able to execute code.

Windows - Since XP SP2.

Hardware mechanism. Both Intel and AMD implemented it.

- NX bit. Not active by default. Choice dictated by legacy programs...
- Compatibility problems. PAE mode needed.

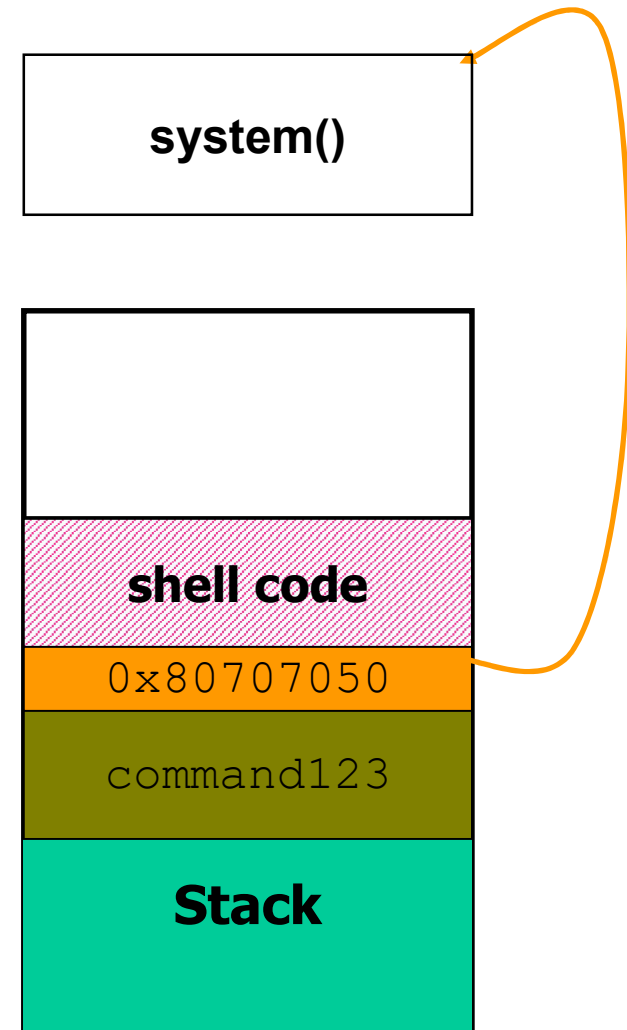
Can also be enforced purely in software (cf. Linux PaX).

“Backdoors” and Security Paranoia

- Any software could contain a backdoor.
- But can we not “audit” the program?
 - we can audit the program source; means close to nothing!
- potentially malicious (or just exploit-able) components:
 - critical part: crypto, modify 3 bits, nobody can tell the difference
 - compiler is malicious and introduces backdoors?
 - compiler side of memory management
 - OS side of memory management (e.g. RAM compression attacks)
 - CPU microcode updates
 - attack can be embedded in SSD firmware
 - new frontier: Intel optane RAM
- Defeating the “trusting trust” attack
 - Importance of “deterministic builds”: several people build the exe and compare the SHA256(final exe).
 - used a lot in crypto currency.
 - example of defence in depth.
- Key paper: Thompson, Ken. "Reflections on trusting trust." Communications of the ACM 27.8 (1984): 761-763.


DEP Solves The Problem?

- Only prevents injection of code.
- the **'return-to-libc' exploit**:
The hacker can overwrite not with code but with a system call plus parameters that will contain the instructions for the shell (!!!).
 - System("command123")
- Details depend a lot on OS.
 - this attack works for simple OS with monolithic kernel...
 - and in many other cases
- Calling the OS functions by direct jumps is not a secure practice...
 - old attack, many changes since...



Gadgets

- Def: a **gadget** is a part of legitimate exe which ends with RET.
- **x86 instructions are NOT aligned!**



```
f7c707000000f9545c3 → test edi, 0x7 ; setnz byte ptr [rbp-0x3d] ;  
c707000000f9545c3 → mov dword ptr [rdi], 0xf000000 ; xchg ebp, eax ; ret
```

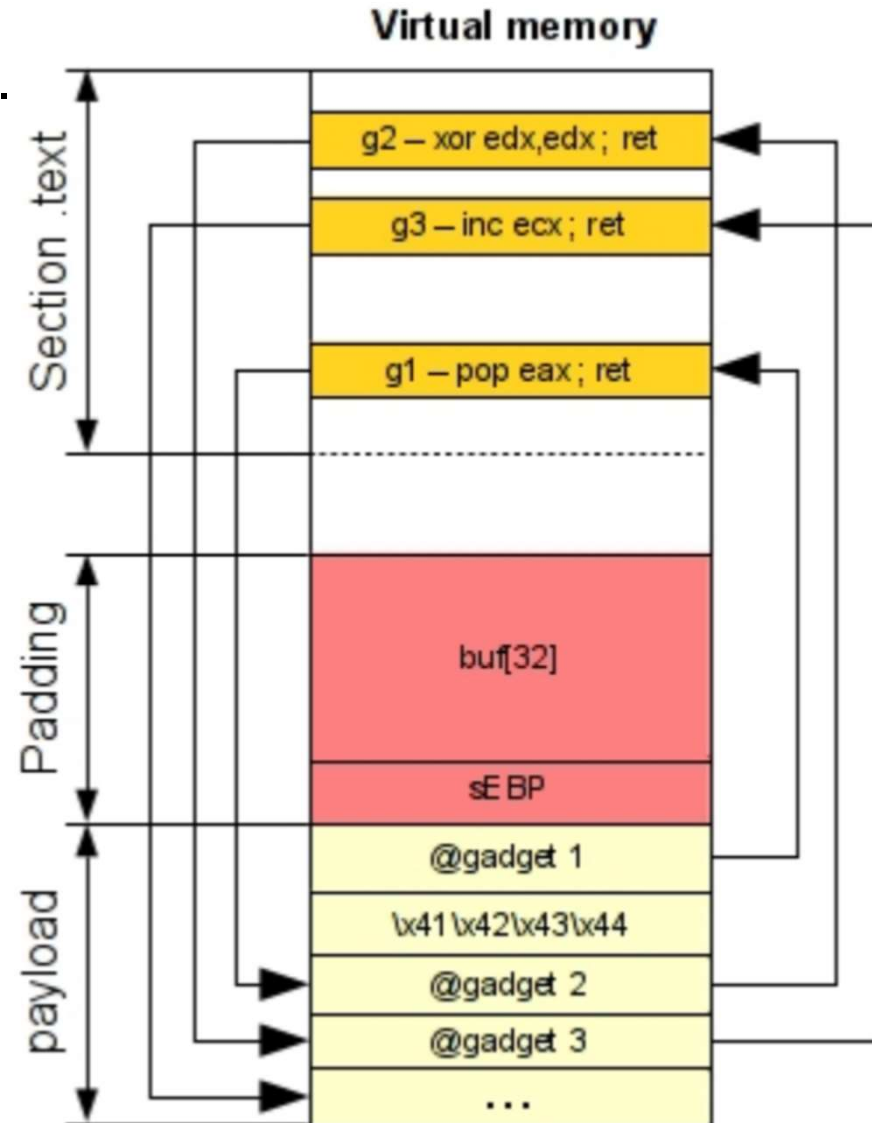
Gadgets

- Def: a **gadget** is a part of legitimate exe which ends with RET.
- x86 instructions are NOT aligned!**

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns
- And so on until all instructions that you want are executed

the real execution is:

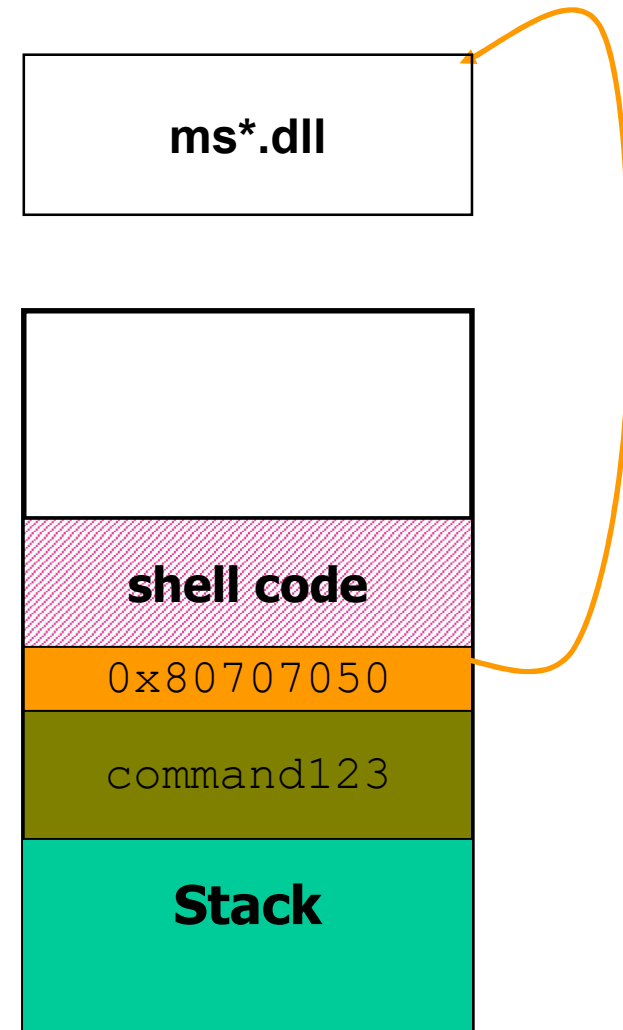
pop	eax
xor	edx, edx
inc	ecx



Preventing Attacks on System Calls

details depend a lot on OS...

- Can we prevent the 'return-to-libc' exploits with Windows dlls?
- Answer: In Windows, at boot time the order and location of system calls **WILL be randomised**.
- Lowers considerably the chances to succeed, (does not eliminate the attack)



DEP Solves The Problem?

- Can still jump to some code injected on the heap
- Does not prevent against attacks on the heap... see later slides.

Input Validation

- Application-specific: check if intended length and format.
 - use special encoding for inputs
 - use encrypted inputs, check length
 - the attack is unlikely to do anything intended?
 - If stream cipher, can flip bits to change one character...
- Routines that remove dangerous characters.
 - In PHP, using the `htmlspecialchars()` function.
 - In an SQL request, use `mysql_real_escape_string()`

C Tips – Replace by

`sprintf(buf, ...)` `snprintf(buf, buflen, ...),`

`scanf("%s", buf)` `scanf("%10s", buf),`

`strcpy(buf, input)` `strncpy(buf, input, 256)`

etc...

Solutions (4)

Automated tools working on:

Source code:

These find lots of bugs, but not all.

Ready exe:

- Taintcheck: fix ready exe files...

Solutions (5)

Replacement libraries:

Example: libsafe – dynamically linked library, will intercept calls to strcpy and check buffer sizes..

StackShield – an assembler file processor for GCC

- keeps backup copies of SFP and RET at the beginning of local variables,
- compare before exiting the function.

Solutions (6)

Instruction Set Randomization (ISR) – runtime encryption of CPU instructions... different for each program, makes code injection impossible.

Heap Overflow Attacks

(about chained lists pointers etc)

Insights

How Memory Management is implemented?

(harder to design a working attack, less standard than stack attacks...)

Implemented by a compiler through its standard dynamic libraries, example: `msvc*.dll` that contain executable already compiled functions.

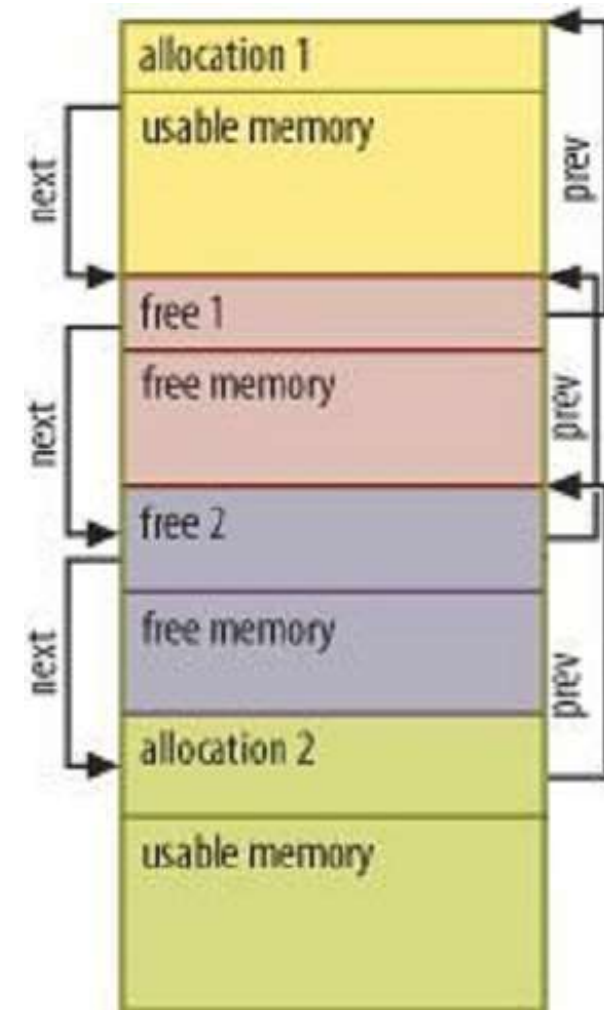
Main idea: the design of these memory management routines can be exploited. How? A bit complex.

Insights

Heap managers have linked lists with forward/backward pointers, sizes, and data fields.

```
struct HeapBlockHeader
{
    HeapBlockHeader* next;
    HeapBlockHeader* prev;
    int size;

    // Actual heap buffer follows this structure.
};
```



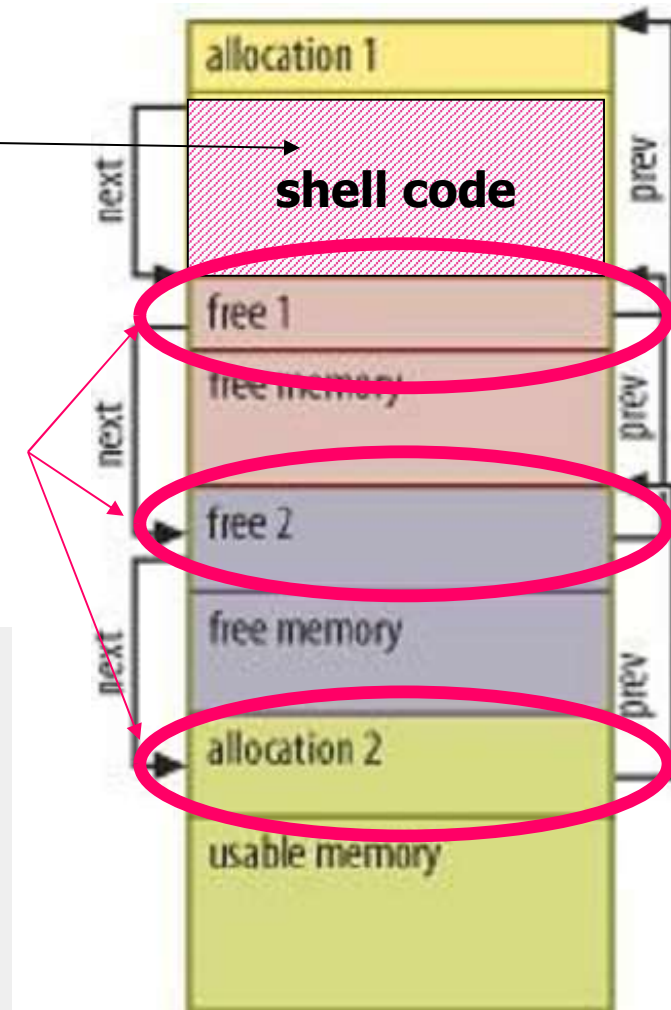
What the attacker can do?

A simple buffer overrun (works only forwards):

- can contain code chosen by the attacker.
- plus extra bytes that will overwrite the “**malloc** meta data” for the next 3 blocks
 - = the **prev/next** pointers in these blocks,
 - overwritten by values chosen by the attacker...

```
struct HeapBlockHeader
{
    HeapBlockHeader* next;
    HeapBlockHeader* prev;
    int size;

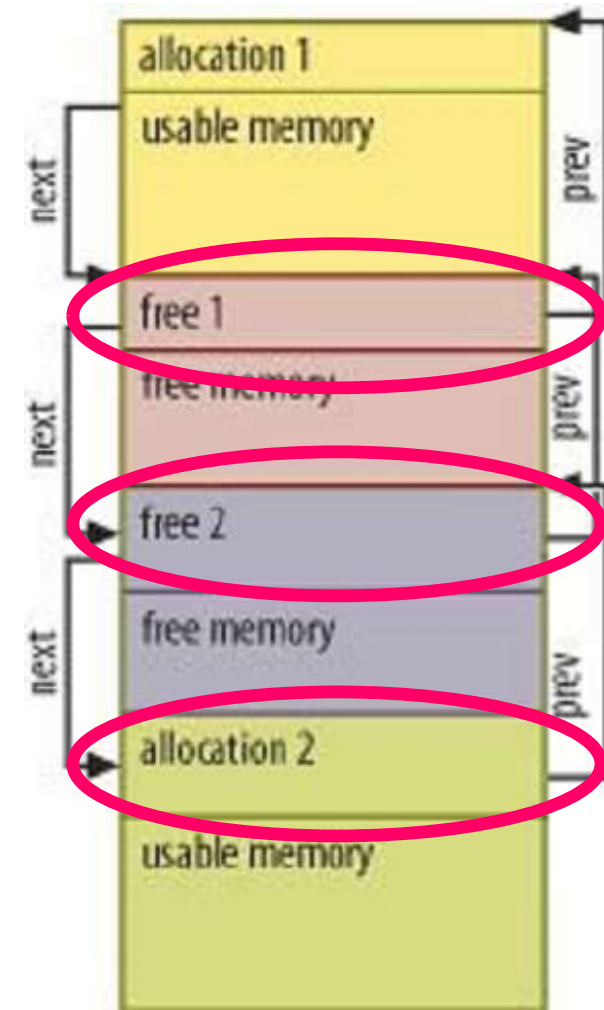
    // Actual heap buffer follows this structure.
};
```



What the attacker can do?

What happens when the routine freeing the memory is called?

On this picture, allocations 1 and 2 are already freed, which maybe happens a bit later during the same function call... The next step is to merge these two free blocks. Why?



Concatenation after free()

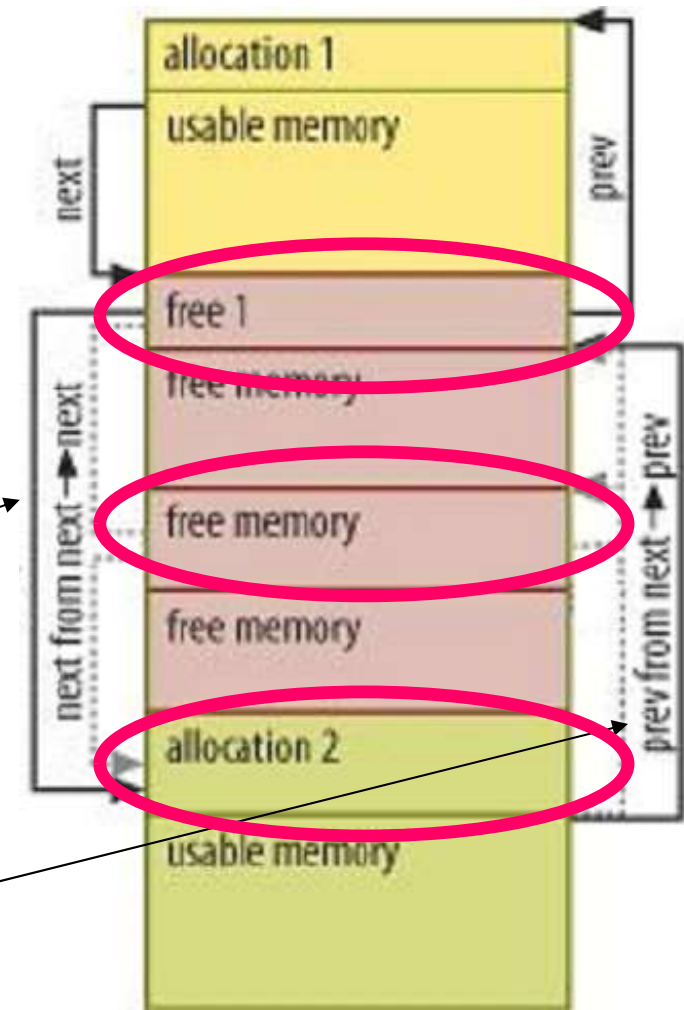
Defragmentation is important:

otherwise allocation of large blocks might fail and the program would terminate with an out of memory message though there is plenty of memory left...

This mechanism is typically automatic and sometimes is also done with a certain delay, but frequently may or will be called before the current C or C++ function exits...

$\text{hdr} \rightarrow \text{next} = \text{hdr} \rightarrow \text{next} \rightarrow \text{next}$

$\text{hdr} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{prev} = \text{hdr} \rightarrow \text{next} \rightarrow \text{prev}$



Insights

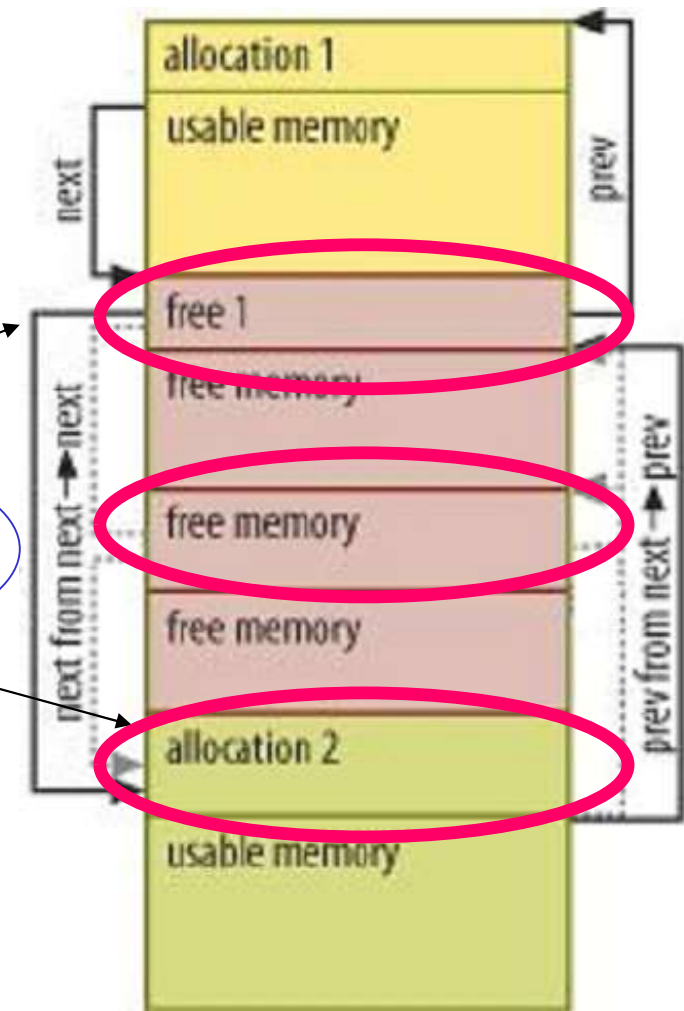
In heap attacks none of these addresses will ever be used as jump address. Seems hopeless?

It is more subtle than that. What we do is to overwrite a return address elsewhere. On the stack. By abusing this specific “defragmentation” method/routine, when it is called (immediately or later).

The attacker can control both:

- the address where a certain pointer will be written automatically by the heap Mgmnt
- the value of this pointer to be overwritten

$\text{hdr} \rightarrow \text{next} =$
 $\text{hdr} \rightarrow \text{next} \rightarrow \text{next}$



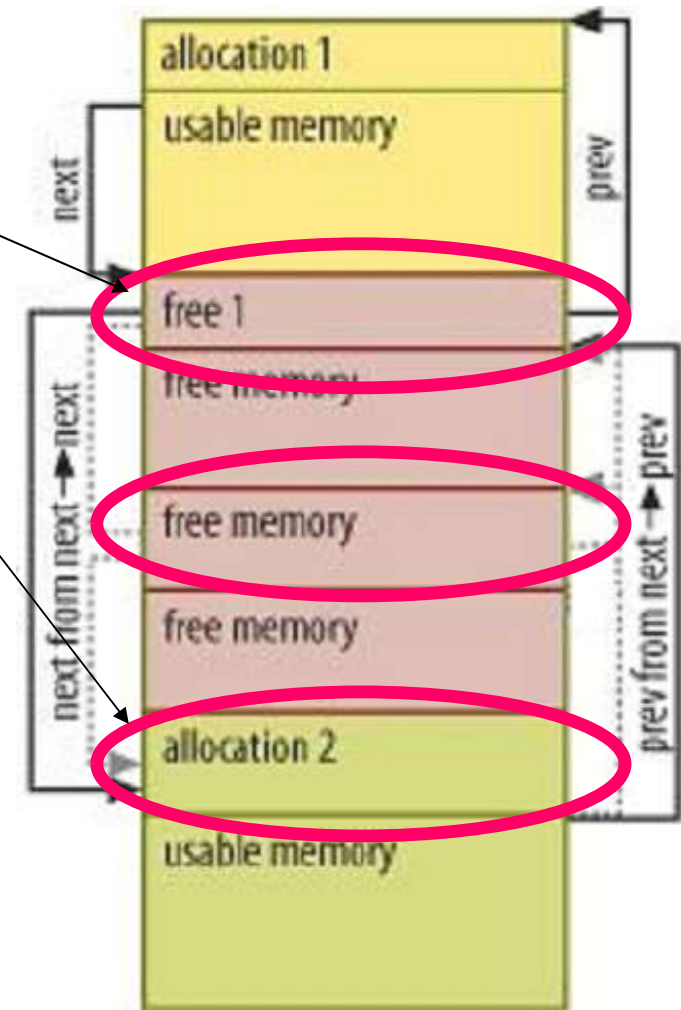
Insights

Suppose I override these links to point

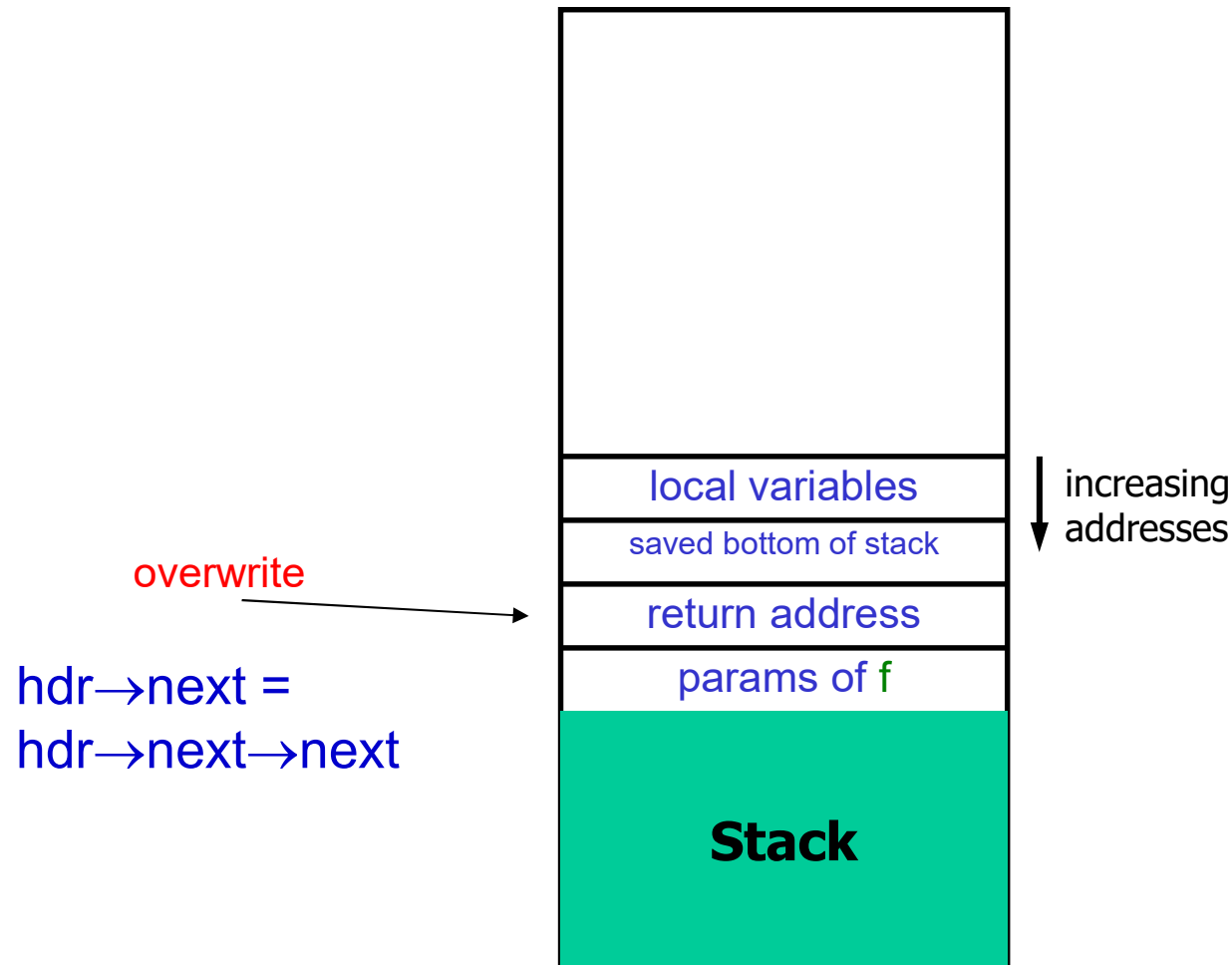
- `hdr→next` = to the return address of the function on the stack.
- `hdr→next→next` = a pointer to code (probably just in the buffer I overran)

When the heap manager merges the two blocks, it will actually overwrite the return address on the stack with a pointer to code I control.

This will be called after the current function exits.



exploit on f



exploit on f

