

CE 874 - Secure Software Systems

Lecture 1

Mehdi Kharrazi

Department of Computer Engineering
Sharif University of Technology



Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.



Software Faults

- Software are developed by humans and therefore are not perfect
- A human error may introduce a bug (or fault)
- Are all software faults security bugs?



Software Insecurity

- A software bug or software fault may be a security bug or vulnerability
 - When the bug is triggered or exploited it compromises the security of the software system



Software Security

- Easy, just write perfect software!
 - Is that actually enough?
- Easy, just write perfect software and have perfect users!
 - Is that actually enough?
- Easy, just write perfect software, have perfect users, and configure software perfectly!
 - Is that actually enough?
- Easy, just write perfect software, have perfect users, configure software perfectly, and use a perfect Operating System!



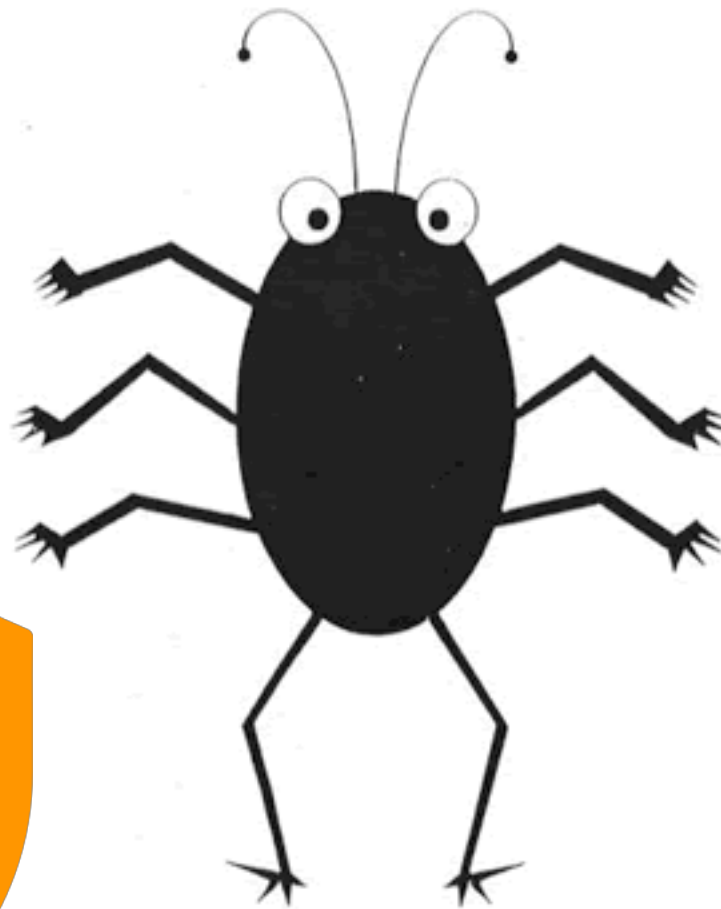
Software Security

- Easy, just write perfect software, have perfect users, configure software perfectly, use a perfect Operating System, use a perfect hypervisor, run on a system with perfect firmware, run on a system with perfect hardware, ...

Really depend on how you look at it



MELTDOWN



BUG



FEATURE

**Speculative
Execution**



Examples (CVE- 2009-4307)

```
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
```

/ There are some situations, after shift the value of 'groups_per_flex' can become zero and division with 0 will result in fixpoint divide exception */*

```
if (groups_per_flex == 0)
```

```
    return 1;
```

```
flex_group_count = ... / groups_per_flex;
```

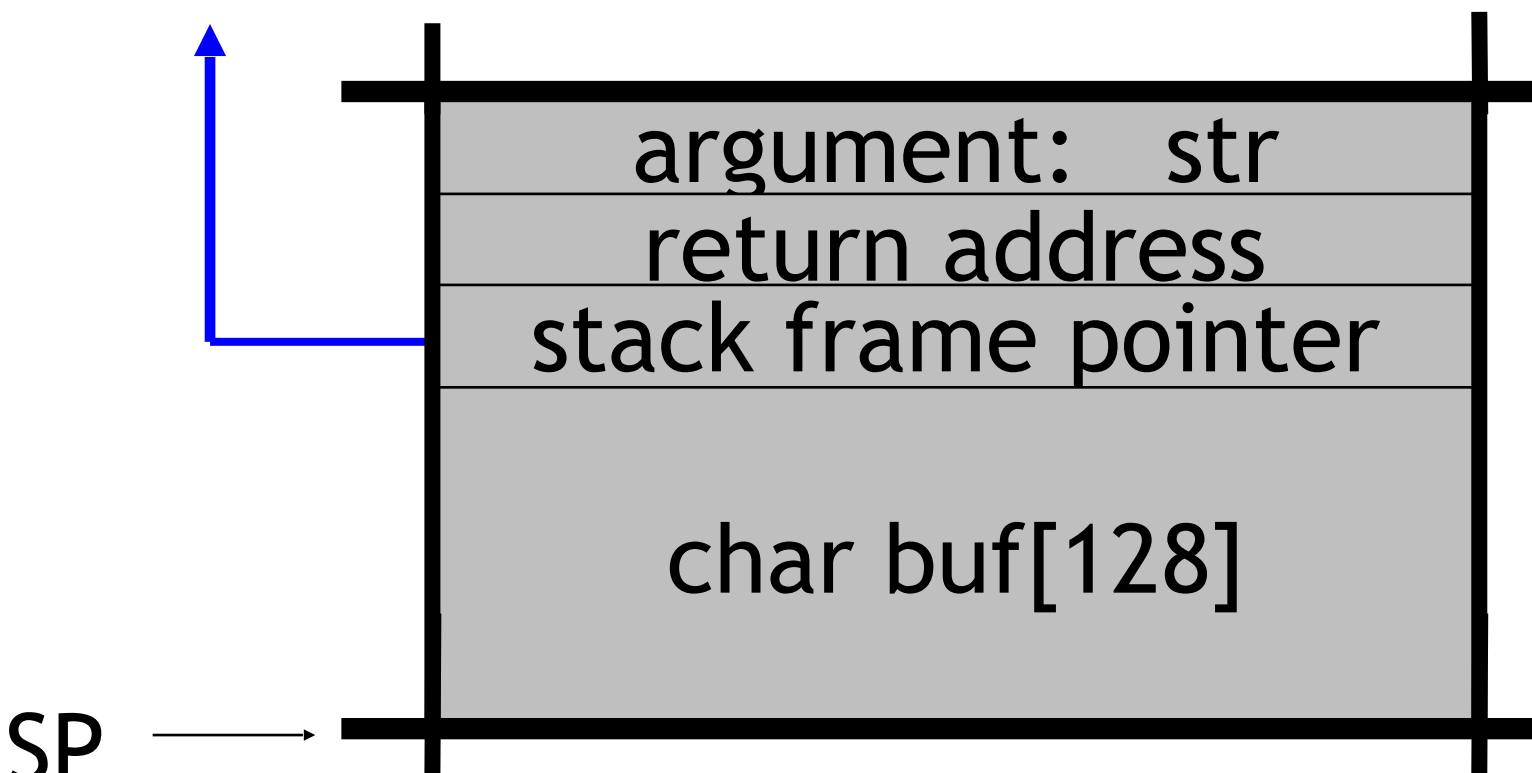
- X86 32bit, shift inst. truncates the shift amount to 5 bits. (32 shift becomes 0)
- PowerPC 32bit, shift inst. truncates the shift amount to 6 bits. (32 shift becomes 1)
- In C, shifting an n-bit integer by n or more bits is undefined behavior.
- Compiler thinks, groups_per_flex will never be zero
 - removed the check when compiling to optimize code



Buffer overflow

- Suppose a web server contains a function:
- When func() is called stack looks like:

```
void func(char *str) {  
    char buf[128];  
  
    strcpy(buf, str);  
    do-something(buf);  
}
```

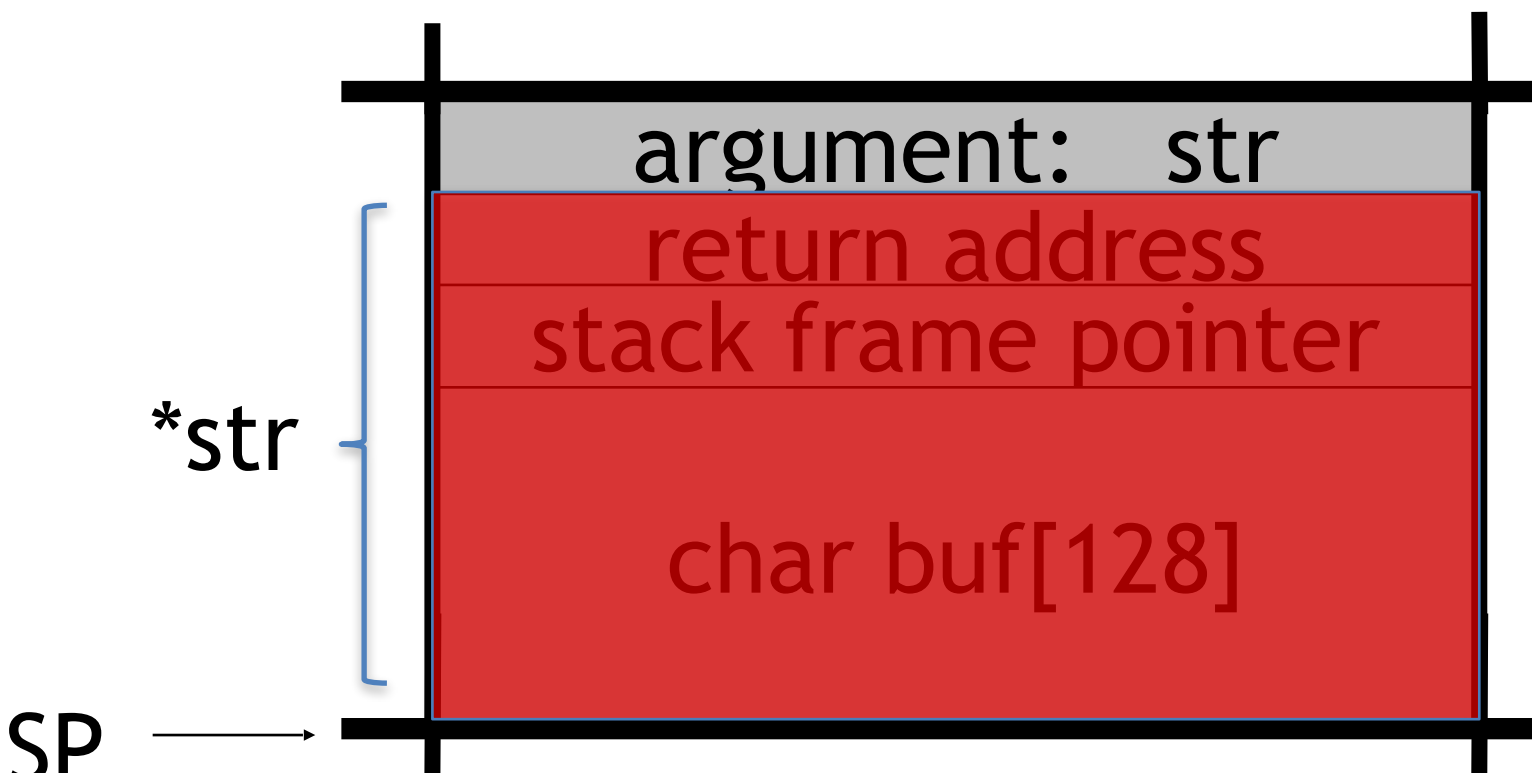




Buffer overflow

- What if `*str` is 136 bytes long?
- After `strcpy`:

```
void func(char *str) {  
    char buf[128];  
  
    strcpy(buf, str);  
    do-something(buf);  
}
```

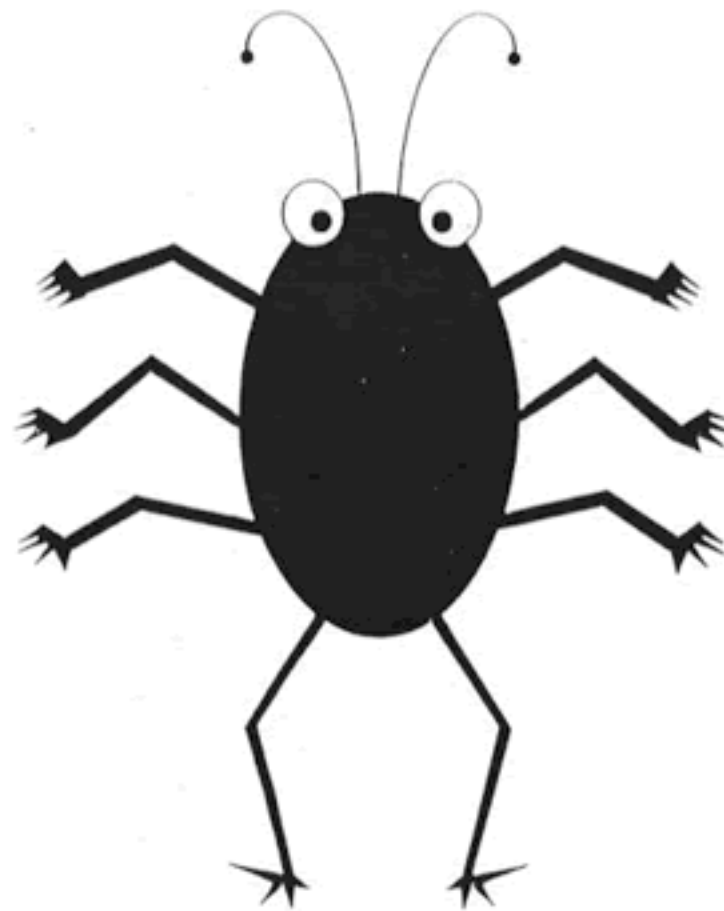


Problem:
no length checking in `strcpy()`

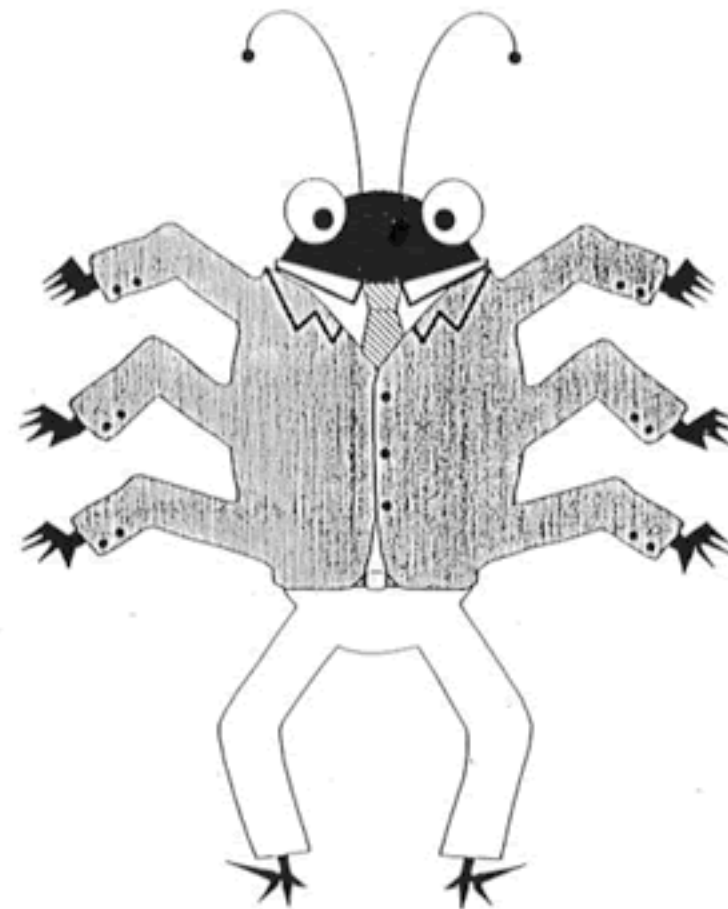


Other Examples

- Out of bound memory access
- Temporal Memory Safety Violations
- Integer overflow
-



BUG



FEATURE



Vulnerabilities

HeartBleed

CVE-2014-0160

Affected over 600,000 websites



Shellshock

CVE-2014-6271

The impact is anywhere from 20 to 50% of global servers



Shellshock

Dirty COW

CVE-2016-5195

Affects all Linux-based operating systems including Android



DIRTY COW

VNOM

CVE-2015-3456

Affected all version of XEN and KVM



VENOM

glib GHOST

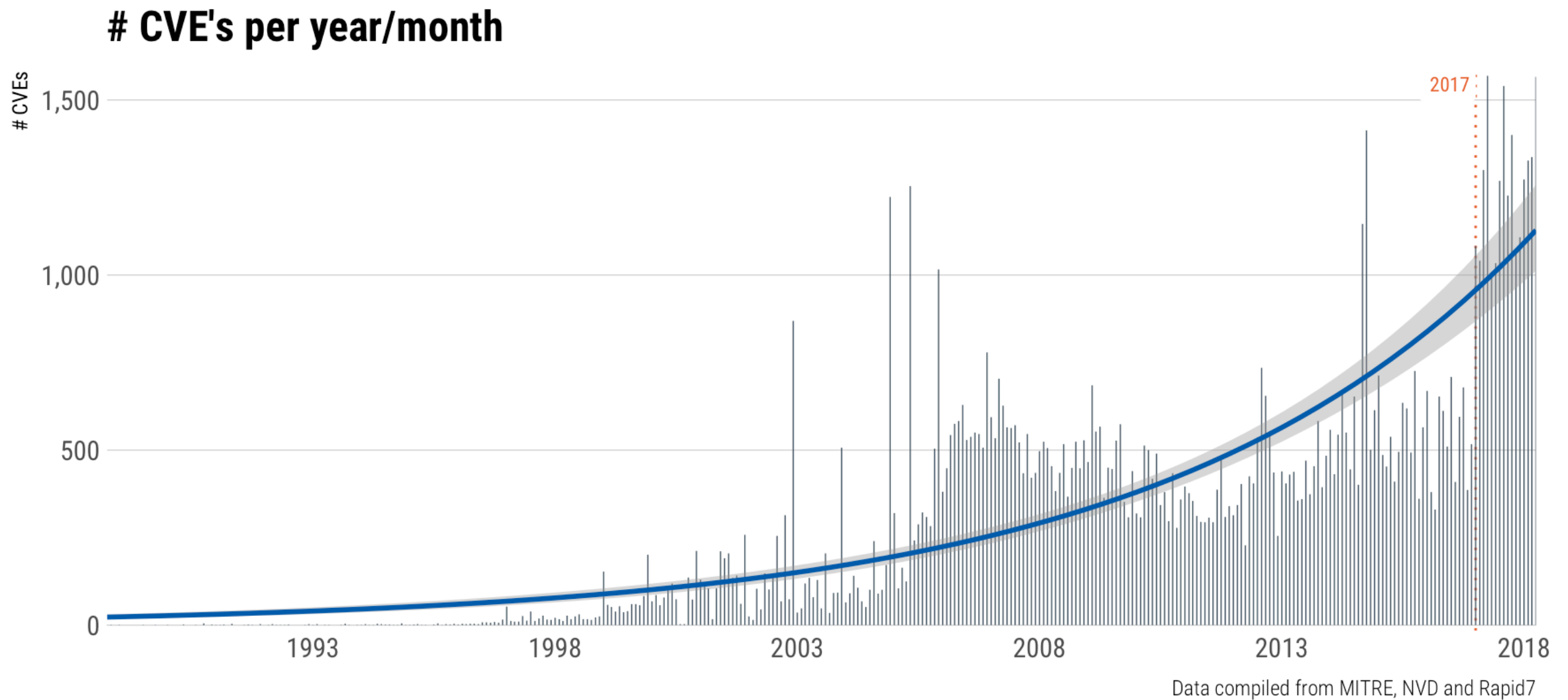
CVE-2015-0235

A core component used in most Linux distributions





CVE Growth



Who cares if there are vulnerabilities???

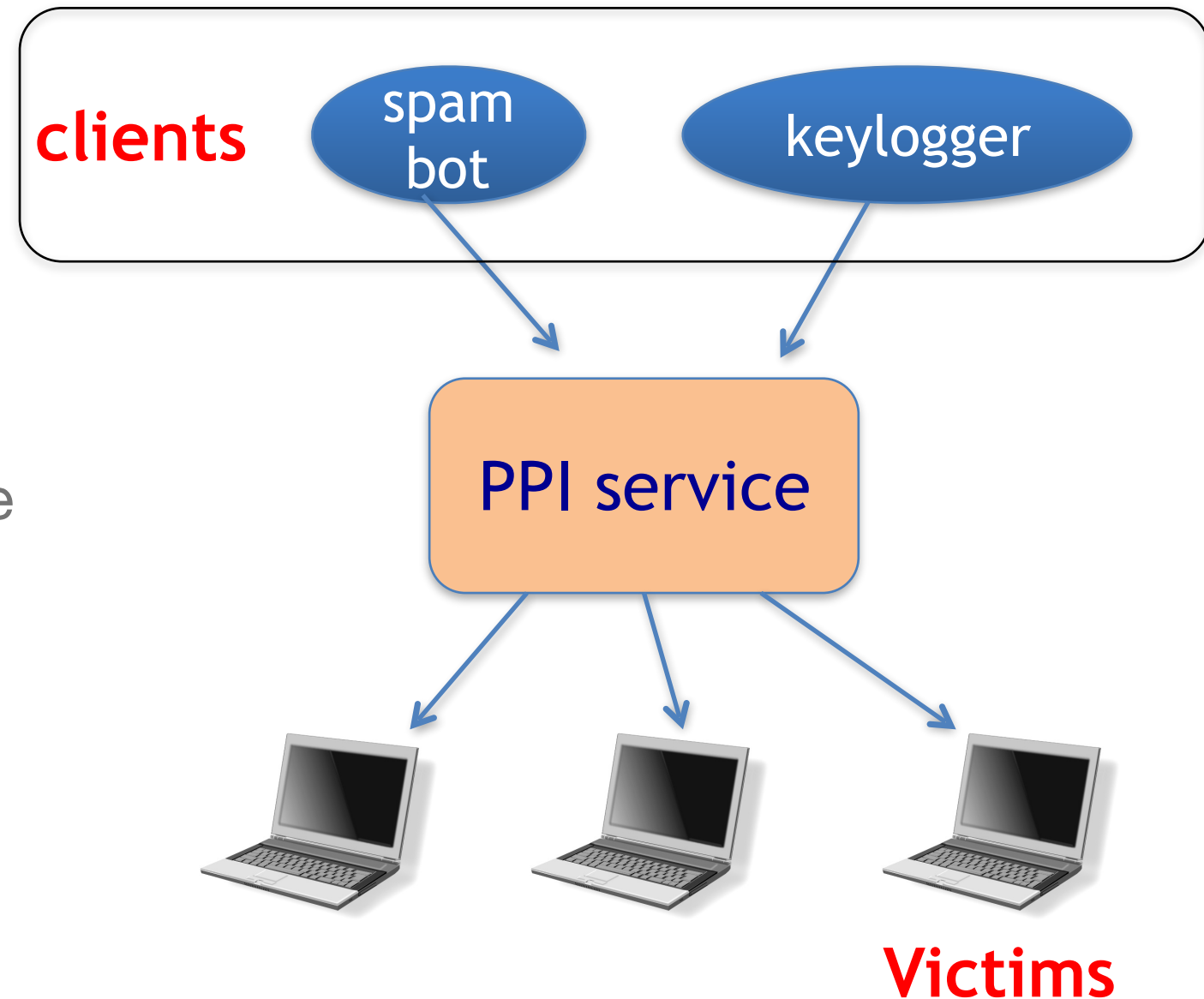


Marketplace for owned machines

Pay-per-install (PPI) services

PPI operation:

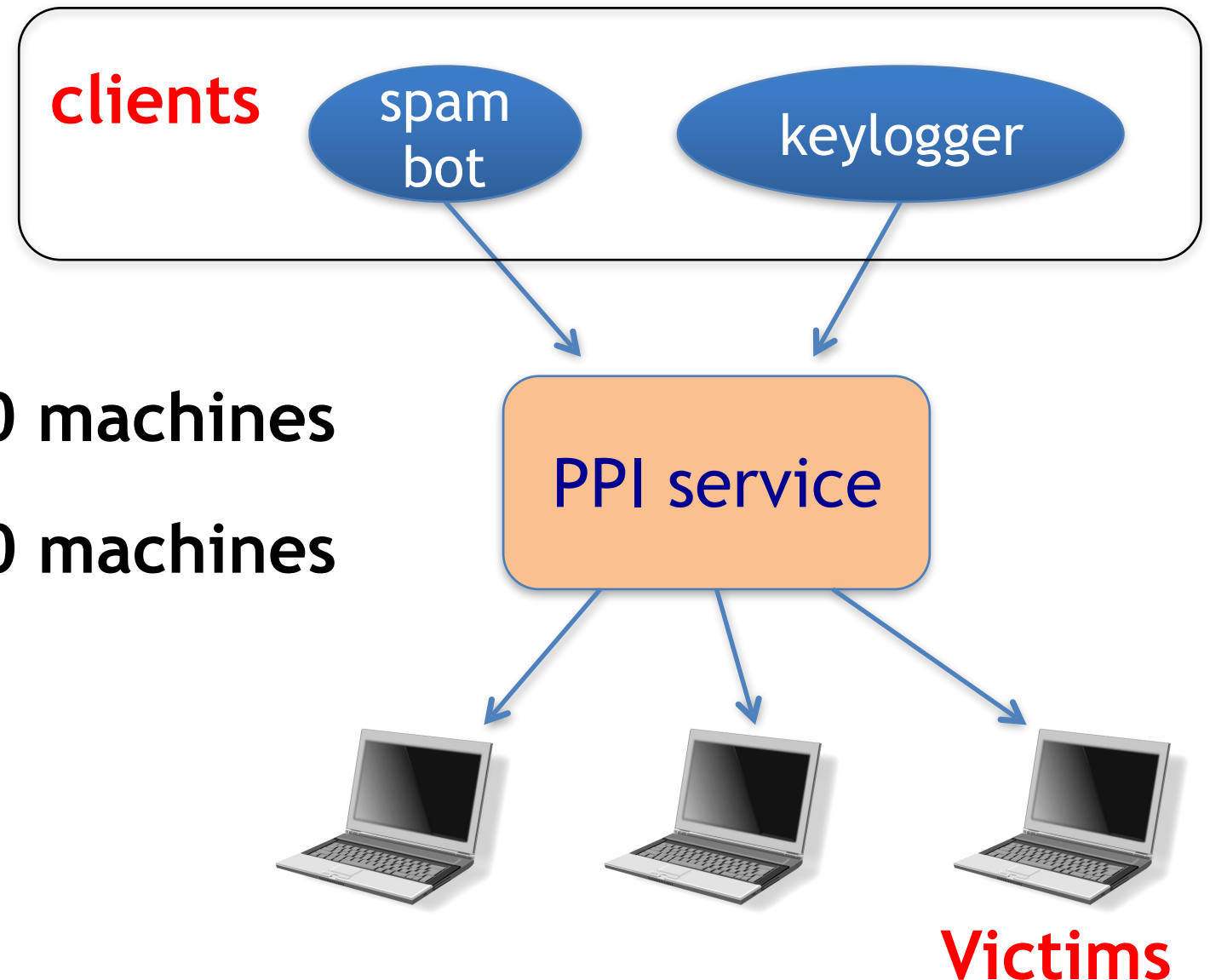
1. Own victim's machine
2. Download and install client's code
3. Charge client



Source: Cabalero et al. (www.icir.org/vern/papers/ppi-usesec11.pdf)



Marketplace for owned machines



Cost: US - 100-180\$ / 1000 machines

Asia - 7-8\$ / 1000 machines

Source: Cabalero et al. (www.icir.org/vern/papers/ppi-usesec11.pdf)



Marketplace for Vulnerabilities

Option 1: bug bounty programs (many)

- Google Vulnerability Reward Program: up to \$20K
- Microsoft Bounty Program: up to \$100K
- Mozilla Bug Bounty program: \$7500
- Pwn2Own competition: \$15K

Option 2:

- Zero day initiative (ZDI), iDefense: \$2K – \$25K



Example: Mozilla

Novel vulnerability and exploit, new form of exploitation or an exceptional vulnerability	High quality bug report with clearly exploitable critical vulnerability ₁	High quality bug report of a critical or high vulnerability ₂	Minimum for a high or critical vulnerability ₃	Medium vulnerability
\$10,000+	\$7,500	\$5,000	\$3,000	\$500 - \$2500



Marketplace for Vulnerabilities

Option 3: black market

ADOBE READER	\$5,000-\$30,000
MAC OSX	\$20,000-\$50,000
ANDROID	\$30,000-\$60,000
FLASH OR JAVA BROWSER PLUG-INS	\$40,000-\$100,000
MICROSOFT WORD	\$50,000-\$100,000
WINDOWS	\$60,000-\$120,000
FIREFOX OR SAFARI	\$60,000-\$150,000
CHROME OR INTERNET EXPLORER	\$80,000-\$200,000
IOS	\$100,000-\$250,000

Source: Andy Greenberg (Forbes, 3/23/2012)

Ok, Important. How we find them?



Audit it

- How much does it take to audit all available programs?

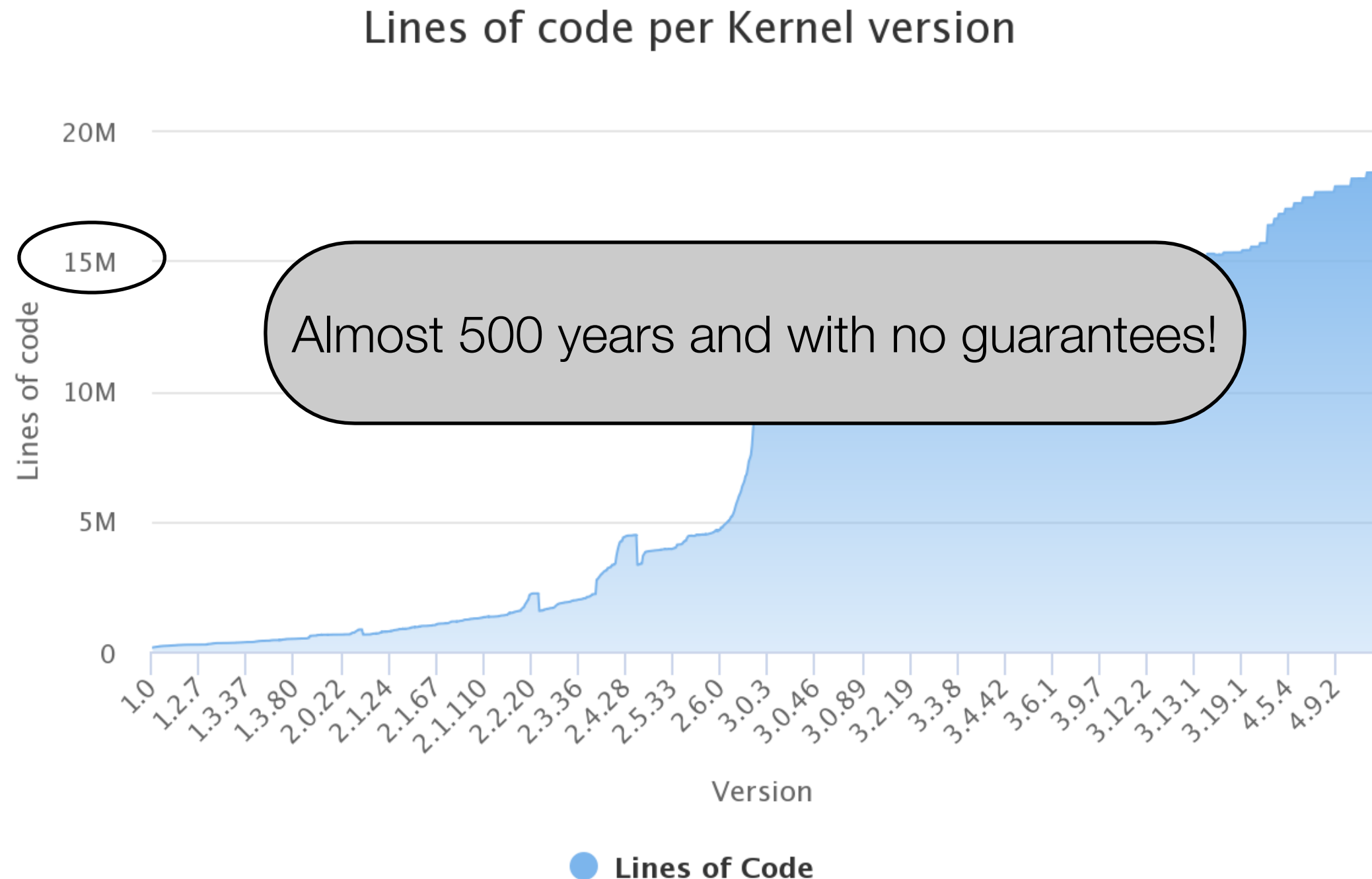
Language	files	blank	comment	code
C	53	12066	3945	46676
C++	28	2027	328	7189
C/C++ Header	114	1775	1351	6891

- It took 2 years to audit TrueCrypt (2013-2015)
- German Government + Cryptographers and Security researchers conducted the audit
- Audit finished April 2015
- CVE-2015-7358 and CVE-2015-7359 discovered September 2015 by Google Zero Project!





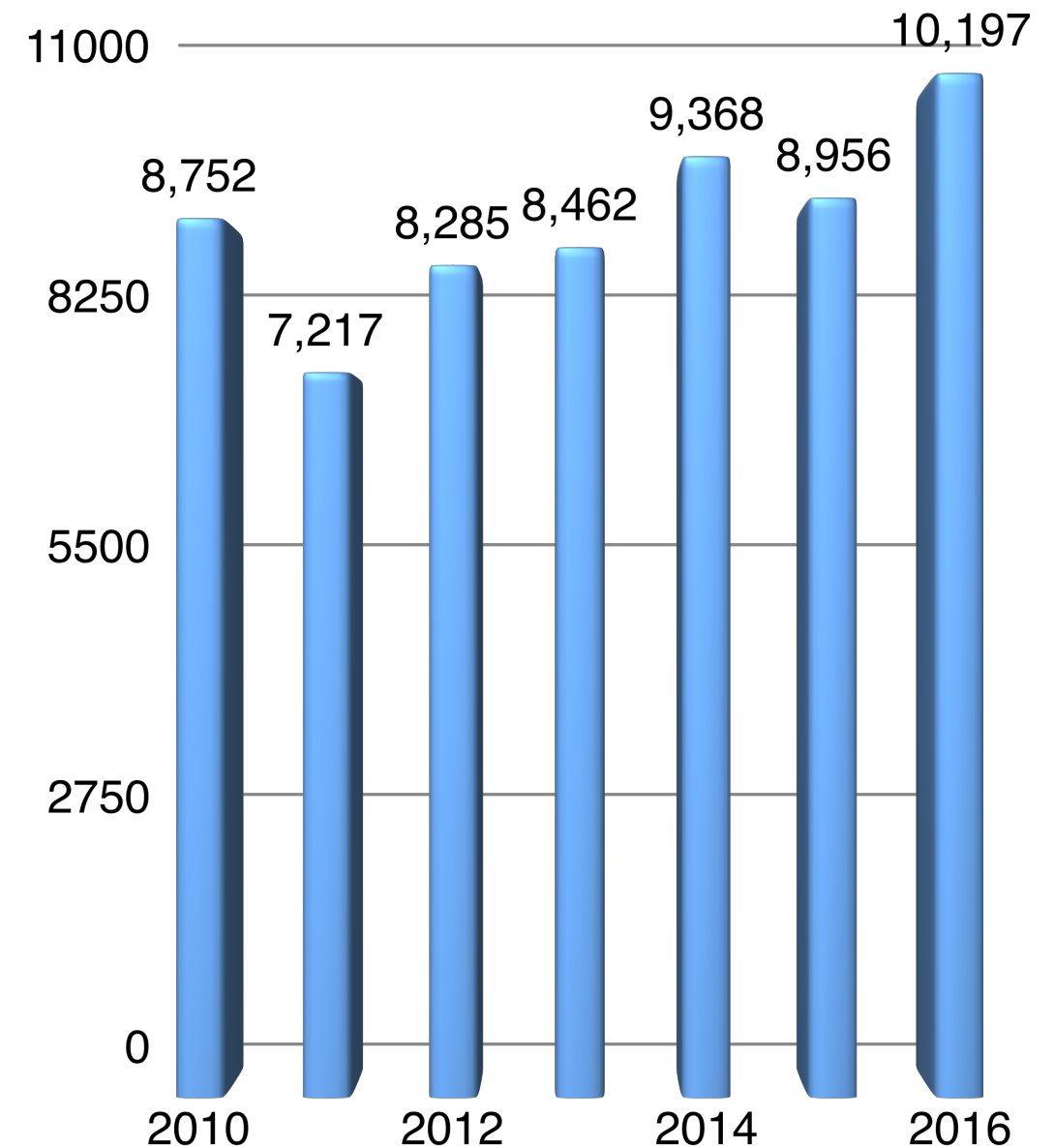
Too much code !





Too much code !!!

- 111 billion lines of new software code is created every year
- Each bug found by hackers first, will lead to a disaster
- Hackers are interested in Exploitable bugs!

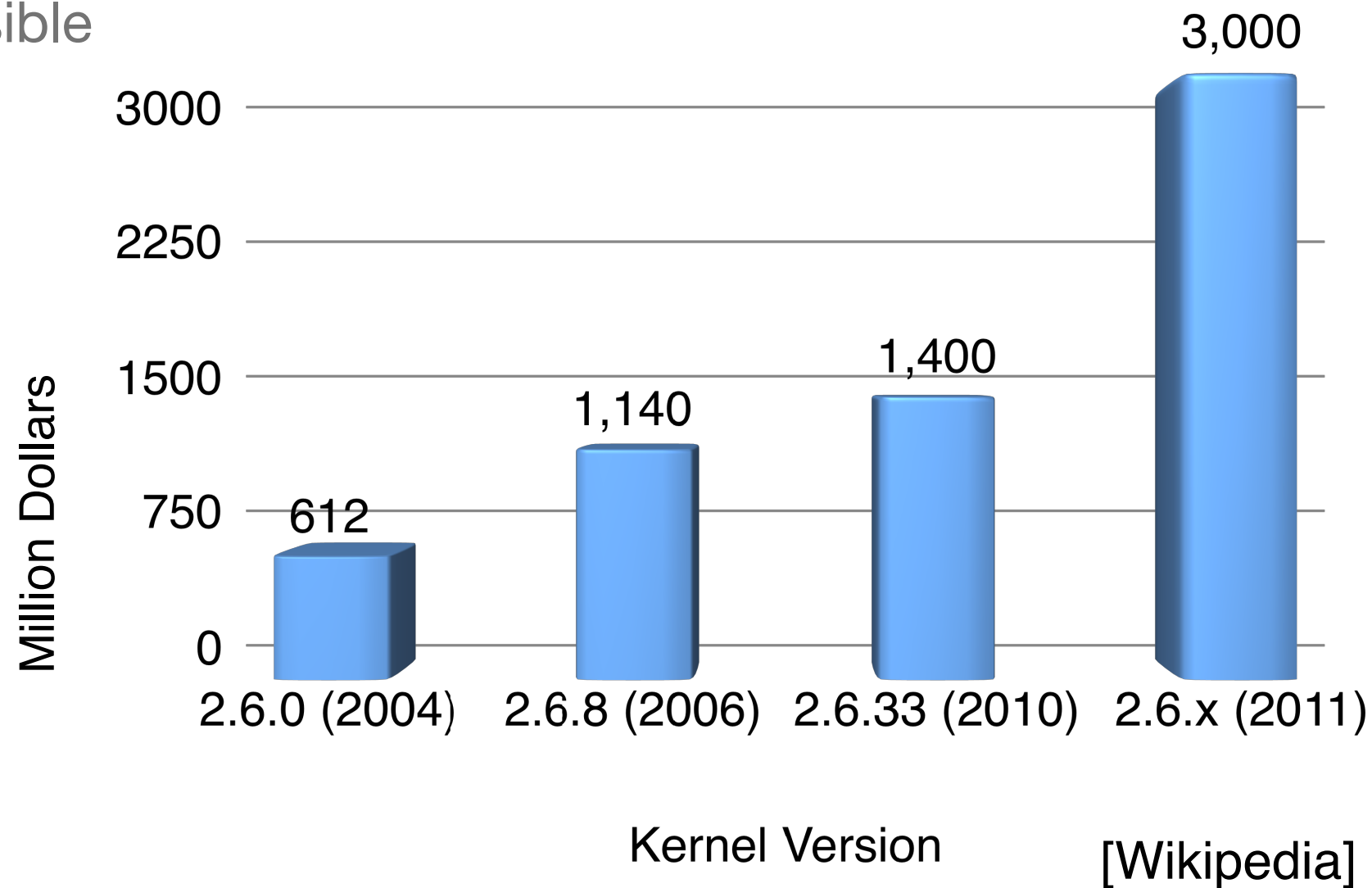


- Number of Vulnerabilities per year; IBM Report 2017



Solutions

- Redevelop Linux Kernel and all other programs
 - Not feasible





DARPA Cyber Grand Challenge

- “Cyber Grand Challenge (CGC) is a contest to build high-performance computers capable of Finding and Fixing Vulnerabilities
- Announced in 2013, and Final Contest held in 2016



- Teams build “Cyber Reasoning Systems” (CRS)
- CRS finds “Proof of Vulnerability” (POV) (automatically exploit)
- CRS fixes vulnerability



Who participated in CGC?



What happens if we don't find them all?



Multiple layers of defense

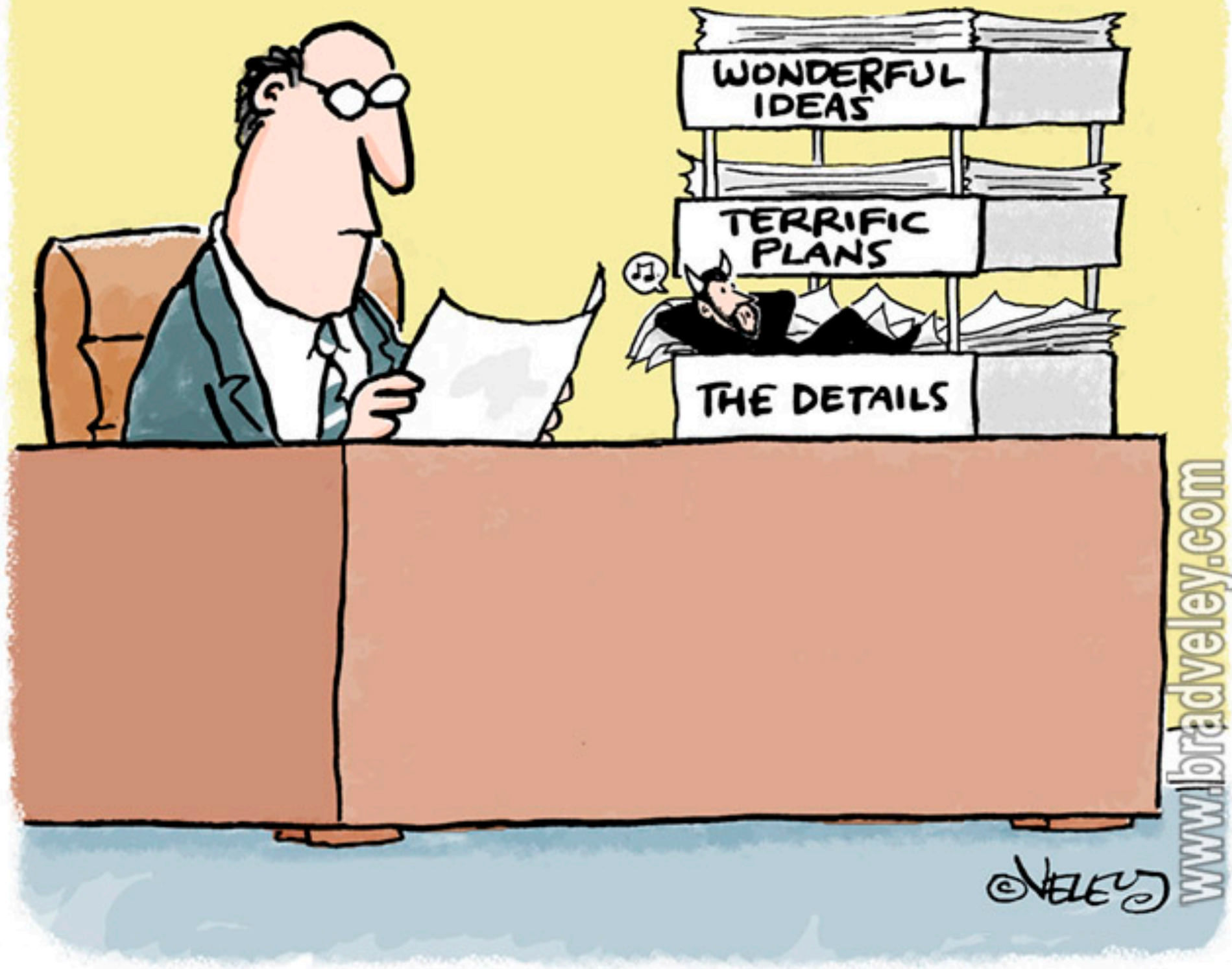
- How to mitigate the vulnerabilities?
 - run-time protection
- How do we look for vulnerabilities?
 - Program analysis
- How do we refrain from one vulnerabilities causing another one?
 - Better Architectures
- How do we refrain from future vulnerabilities?
 - Better programming languages



High level course view

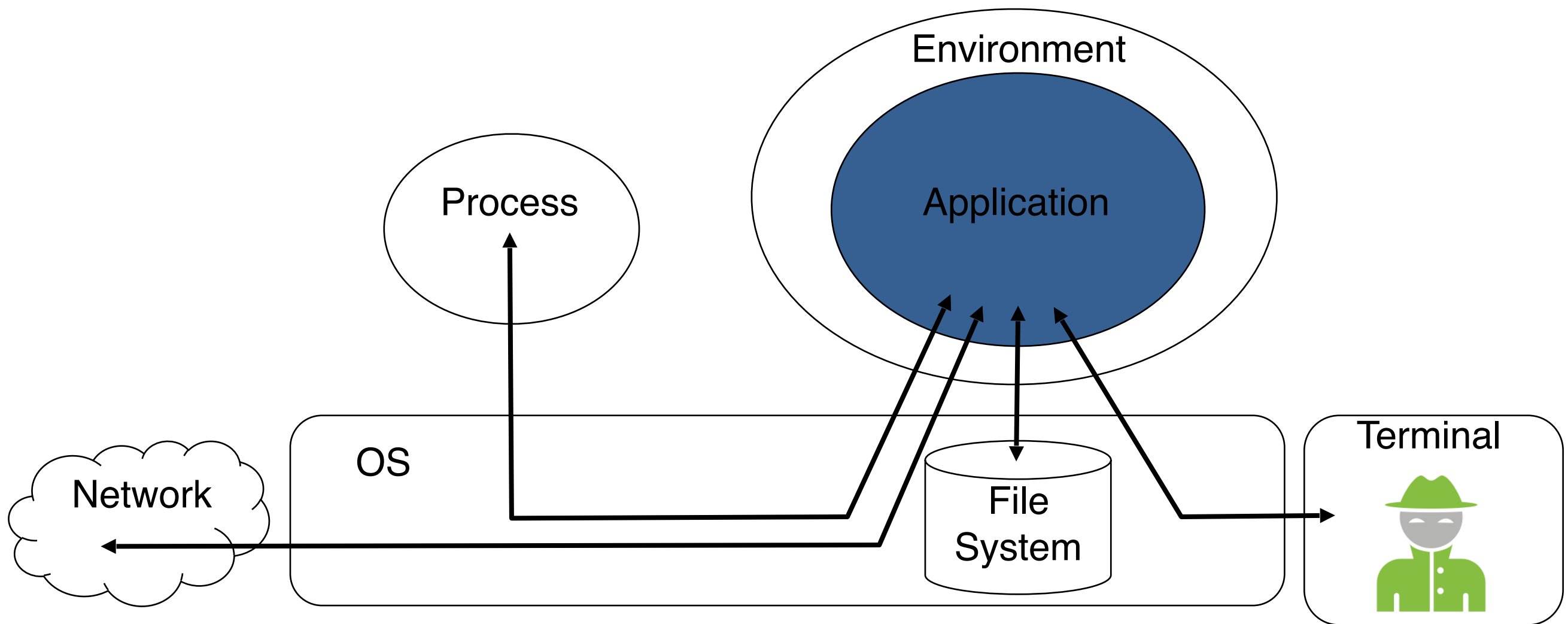
- Classic attacks
 - Buffer Overflow, Format String, ROP, etc.
- Run-time protection
 - Taint tracking, CFI, etc.
- Code analysis
 - Static analysis, Symbolic execution, Fuzzing
- Architecture
 - Sandboxing, VMs, Isolation, Trusted computing
- Web
 - Native client, App isolation
- Language
 - Type systems, WebAssembly

A quick review of some of the very basics!





Application Model





Stages in which there could be a vulnerability

- Design vulnerabilities
 - Flaws in the overall logic of the application
 - Lack of authentication and/or authorization checks
 - Erroneous trust assumptions
- Implementation vulnerabilities
 - Application is not able to correctly handle unexpected events
 - Unexpected input, Unexpected errors/exceptions
 - Unexpected interleaving of events
- Deployment vulnerabilities
 - Incorrect/faulty deployment/configuration of the application
 - Installed with more privileges than the ones it should have
 - Installed on a system that has a faulty security policy and/or mechanism (e.g., a file that should be read-only is actually writeable)



The Life of an Application

- Author writes code in high-level language
- The application is translated in some executable form and saved to a file
 - Interpretation vs. compilation
- The application is loaded in memory
- The application is executed
- The application terminates



Interpretation

- The program is passed to an interpreter
 - The program might be translated into an intermediate representation
 - Python byte-code
- Each instruction is parsed and executed
- In most interpreted languages it is possible to generate and execute code dynamically
 - Bash: `eval <string>`
 - Python: `eval(<string>)`
 - JavaScript: `eval(<string>)`
 - ...



Compilation

- The preprocessor expands the code to include definitions, expand macros
 - GNU/Linux: The C preprocessor is `cpp`
- The compiler turns the code into architecture-specific assembly
 - GNU/Linux: The C compiler is `gcc`
 - `gcc -S prog.c` will generate the assembly
 - Use `gcc`'s `-m32` option to generate 32-bit assembly



Compilation

- The assembler turns the assembly into a binary object
 - GNU/Linux: The assembler is as
 - A binary object contains the binary code and additional metadata
 - Relocation information about things that need to be fixed once the code and the data are loaded into memory
 - Information about the symbols defined by the object file and the symbols that are imported from different objects
 - Debugging information



Compilation

- The linker combines the binary object with libraries, resolving references that the code has to external objects (e.g., functions) and creates the final executable
 - GNU/Linux: The linker is ld
 - Static linking is performed at compile-time
 - Dynamic linking is performed at run-time
- Most common executable formats:
 - GNU/Linux: ELF
 - Windows: PE

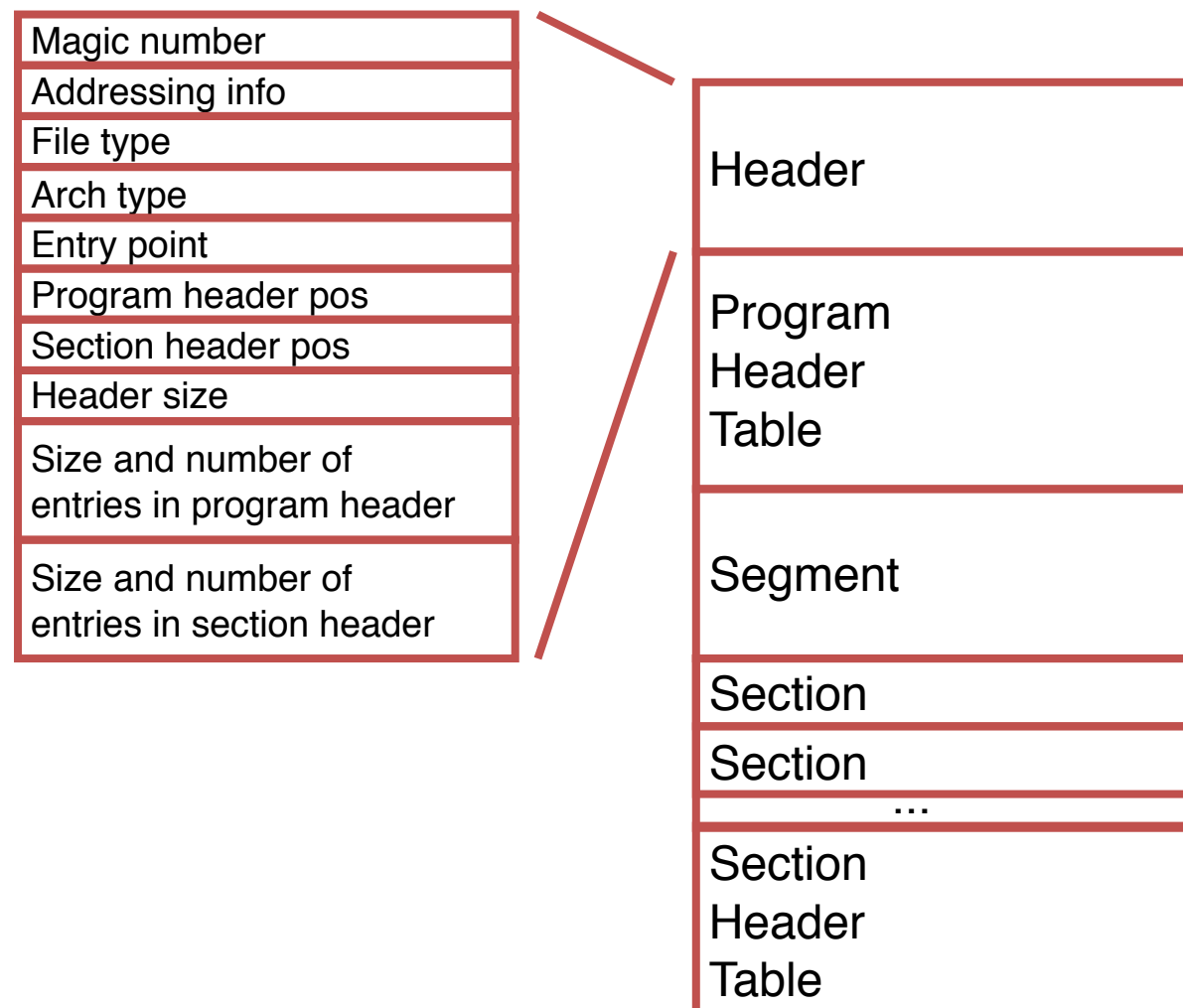


The ELF File Format

- The Executable and Linkable Format (ELF) is one of the most widely-used binary object formats
- ELF is architecture-independent
- ELF files are of four types:
 - Relocatable: need to be fixed by the linker before being executed
 - Executable: ready for execution (all symbols have been resolved with the exception of those related to shared libs)
 - Shared: shared libraries with the appropriate linking information
 - Core: core dumps created when a program terminated with a fault
- Tools: readelf, file



The ELF File Format



- A program is seen as a collection of segments by the loader and as a collection of sections by the compiler/linker
- A segment is usually made of several sections
- The segment structure is defined in the Program Header Table
- The section structure is defined in the Section Header Table



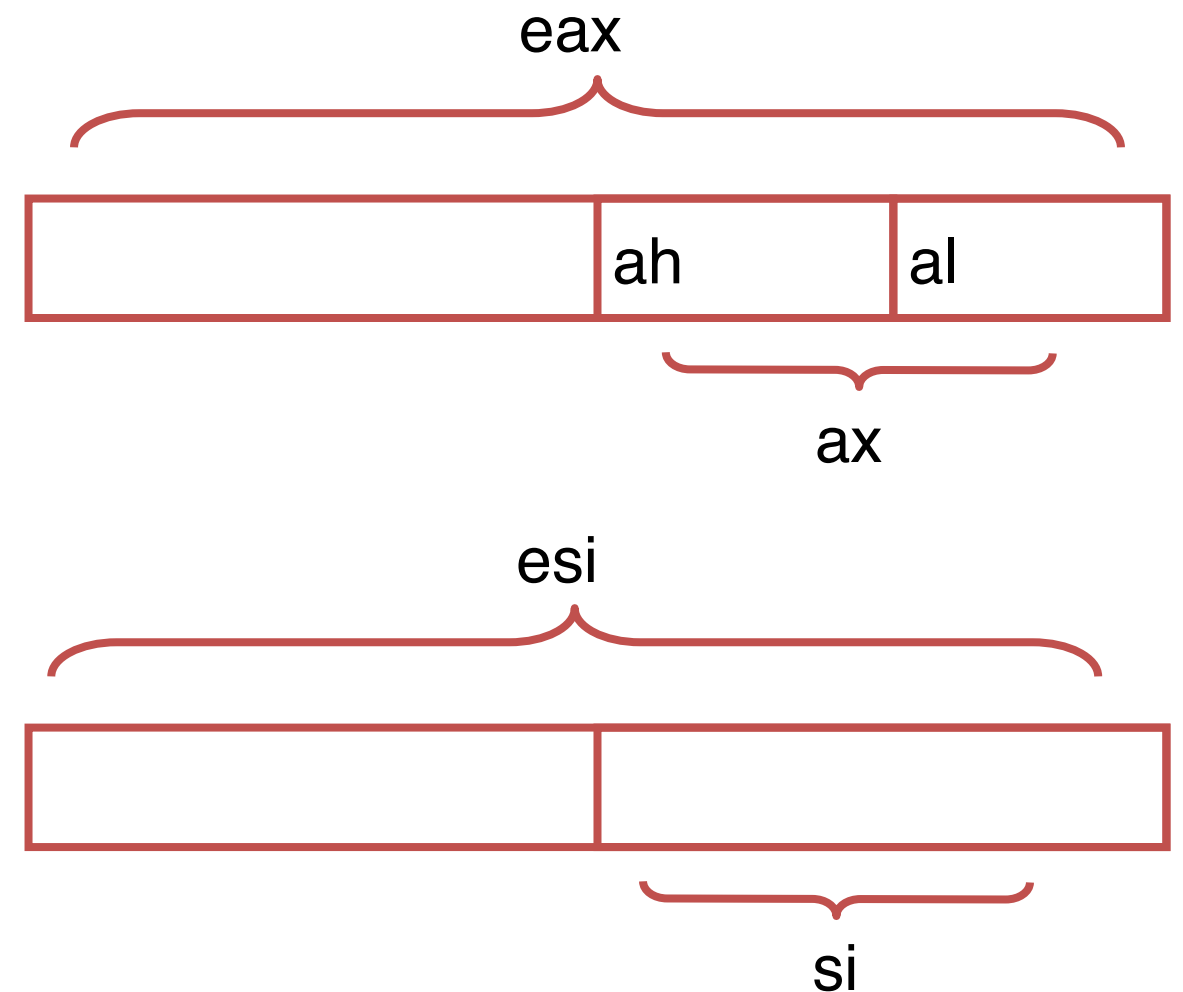
The PE File Format

- The PE file was introduced to allow MS-DOS programs to be larger than 64K (limit of .COM format)
- Also known as the “EXE” format
- The header contains a number of relocation entries that are used at loading time to “fix” the addresses (this procedure is called rebasing)
 - Programs are written as if they were always loaded at address 0
 - The program is actually loaded in different points in memory



x86 Registers

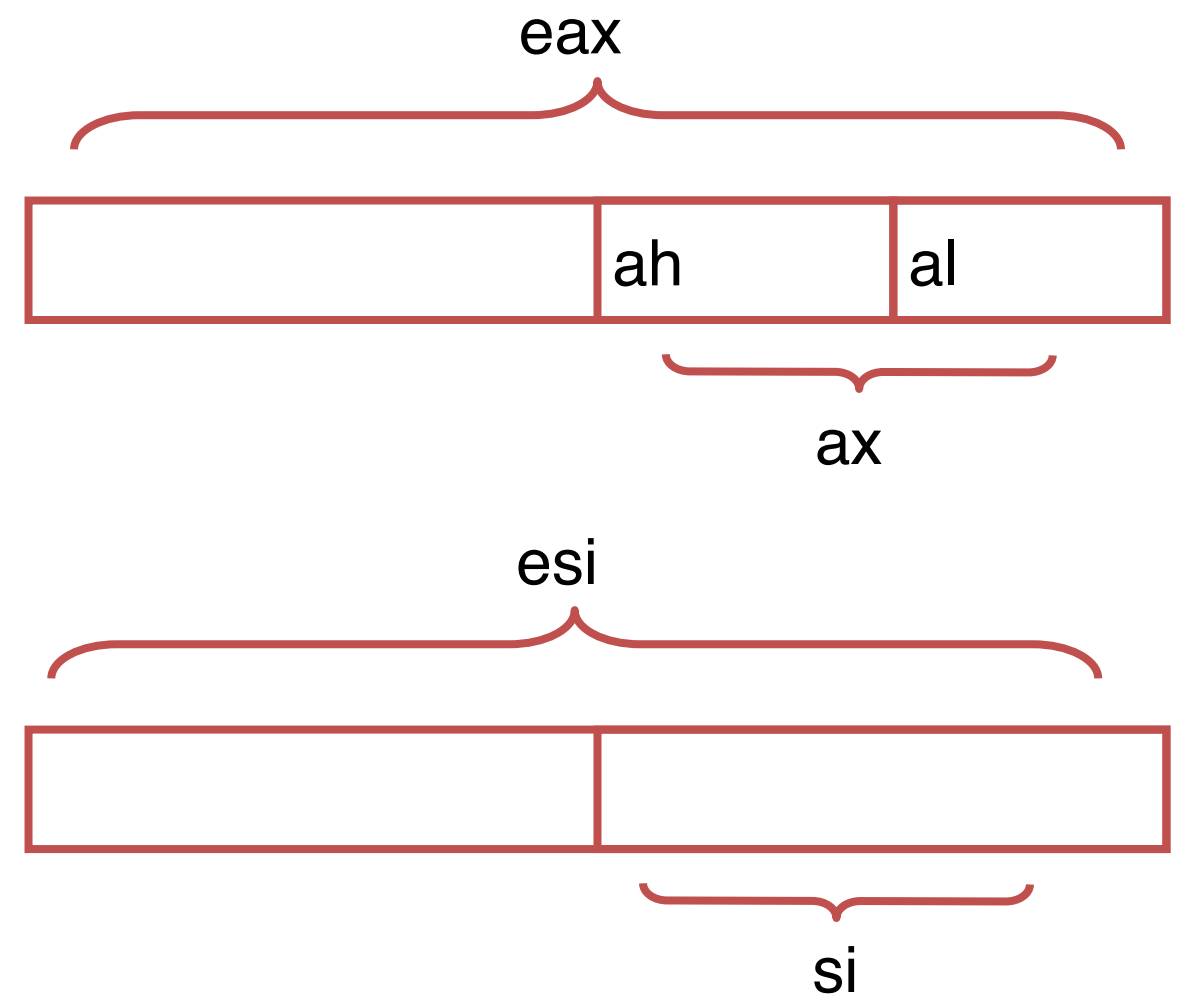
- Registers represent the local variables of the processor
- There are four 32-bit general purpose registers
 - eax/ax, ebx/bx, ecx/cx, edx/cx
- Convention
 - Accumulator: eax
 - Pointer to data: ebx
 - Loop counter: ecx
 - I/O operations: edx





x86 Registers

- Two registers are used for high-speed memory transfer operations
 - esi/si (source), edi/di (destination)
- There are several 32-bit special purpose registers
 - esp/sp: the stack pointer
 - ebp/bp: the frame pointer



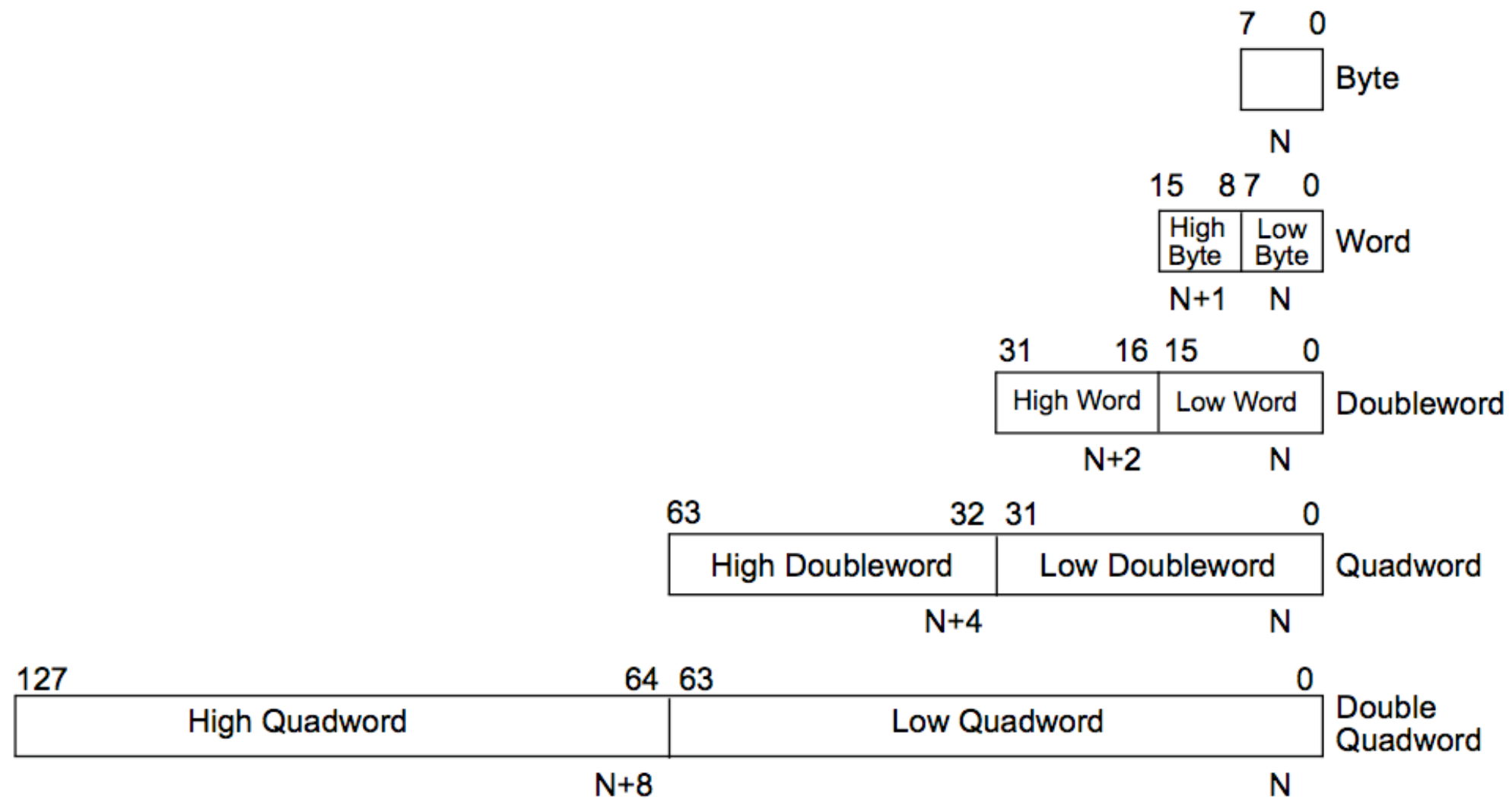


x86 Registers

- Segment registers: cs, ds, ss, es, fs, gs
 - Used to select segments (e.g., code, data, stack)
- Program status and control: eflags
- The instruction pointer: eip
 - Points to the next instruction to be executed
 - Cannot be read or set explicitly
 - It is modified by jump and call/return instructions
 - Can be read by executing a call and checking the value pushed on the stack
- Floating point units and mmx/xmm registers



Data Sizes





x86 Assembly Language

- (Slightly) higher-level language than machine language
- Program is made of:
 - directives: commands for the assembler
 - .data identifies a section with variables
 - instructions: actual operations
 - `jmp 0x08048f3f`
- Two possible syntaxes, with different ordering of the operands!
 - AT&T syntax (objdump, GNU Assembler)
 - mnemonic source, destination
 - DOS/Intel syntax (Microsoft Assembler, Nasm, IDA Pro)
 - mnemonic destination, source
 - In gdb can be set using: `set disassembly-flavor intel/att`



Data Definition

- Constants
 - Hexadecimal numbers start with 0x
- Data objects are defined in a data segment using the syntax
 - `label        type        data1, data2, ...`
- Types can be
 - DB: Byte
 - DW: Word (16 bits)
 - DD: Double word (32 bits)
 - DQ: Quad word (64 bits)
- For example:

```
.data
myvar        DD    0x12345678, 0x23456789        # Two 32-bit values
bar          DW    0x1234                        # 16-bit data object
mystr        DB    "foo", 0                      # Null-terminated string
```




Addressing Memory

- Memory access is composed of width, base, index, scale, and displacement
 - Base: starting address of reference
 - Index: offset from base address
 - Scale: Constant multiplier of index
 - Displacement: Constant base
 - Width: (address suffix)
 - size of reference (b: byte, s: short, w: word, l: long, q: quad)
 - Address = base + index*scale + displacement
 - AT&T Syntax $\rightarrow$ displacement(base, index, scale)
 - Example:
 - `movl -0x20(%eax, %ecx, 4), %edx`



Addressing Memory

- `movl -8(%ebp), %eax`
 - copies the contents of the memory pointed by `ebp - 8` into `eax`
- `movl (%eax), %eax`
 - copies the contents of the memory pointed by `eax` to `eax`
- `movl %eax, (%edx, %ecx, 2)`
 - moves the contents of `eax` into the memory at address $\text{edx} + \text{ecx} * 2$
- `movl $0x804a0e4, %ebx`
 - copies the value `0x804a0e4` into `ebx`
- `movl (0x804a0e4), %eax`
 - copies the content of memory at address `0x804a0e4` into `eax`



Instruction Classes

- Data transfer
 - mov, xchg, push, pop
- Binary arithmetic
 - add, sub, imul, mul, idiv, div, inc, dec
- Logical
 - and, or, xor, not



Instruction Classes

- Control transfer
 - jmp, call, ret, int, iret
 - Values can be compared using the cmp instruction
 - `cmp src, dest #` subtracts src from dest without saving the result
 - Various eflags bits are set accordingly
 - jne (ZF=0), je (ZF=1), jae (CF=0), jge (SF=OF), ...
 - Control transfer can be direct (destination is a constant) or indirect (the destination address is the content of a register)
- Input/output
 - in, out
- Misc
 - nop



Invoking System Calls

- System calls are usually invoked through libraries
- Linux/x86
 - `int 0x80`
 - `eax` contains the system call number



Hello World!

```
.data
hw:
    .string "Hello World\n"
.text
.globl main
main:
    movl    $4,%eax
    movl    $1,%ebx
    movl    $hw,%ecx
    movl    $12,%edx
    int     $0x80
    movl    $0,%eax
    ret
```

```
int main()
{
    printf("Hello, World!");
    return 0;
}

syscall(4, 1, "hello, world!\n", 12);
```



Program Loading and Execution

- When a program is invoked, the operating system creates a process to execute the program
- The ELF file is parsed and parts are copied into memory
 - In Linux `/proc/<pid>/maps` shows the memory layout of a process
- Relocation of objects and reference resolution is performed
- The instruction pointer is set to the location specified as the start address
- Execution begins



Acknowledgments/References (1/2)

- [Adam Doupe] CSE 545, Software Security, Adam Doupe, ASU, Spring 2018
- [Bellovin 06] COMS W4180 — Network Security Class Columbia University.
- [Messmer 08] 10 of the Worst Moments in Network Security History, Events that shock sensibilities and shaped the future, By Ellen Messmer, Network World, 03/11/08
- [Rudis'18] CVE 100K: By The Numbers, Bob Rudis, Apr 30, 2018. <https://blog.rapid7.com/2018/04/30/cve-100k-by-the-numbers/>
- [Mendoza'18] Vulnerabilities reached a historic peak in 2017, Miguel Ángel Mendoza 5 Feb 2018. <https://www.welivesecurity.com/2018/02/05/vulnerabilities-reached-historic-peak-2017/>



Acknowledgments/References (2/2)

- [Wang 2012] Undefined Behavior: What Happened to My Code?, X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, M.F. Kaashoek, APSys '12, 2012.
- [Regehr 2017] Undefined Behavior in 2017, John Regehr blog, 2017.
- [Williams 2017] Meltdown and Spectre - understanding and mitigating the threats, Jake Williams, SANS / Rendition Infosec, 2017 (slides)
- [cs 155] Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University, 2015.