Sicurezza Informatica

1. Got Attack
2. Defending against buffer overflows

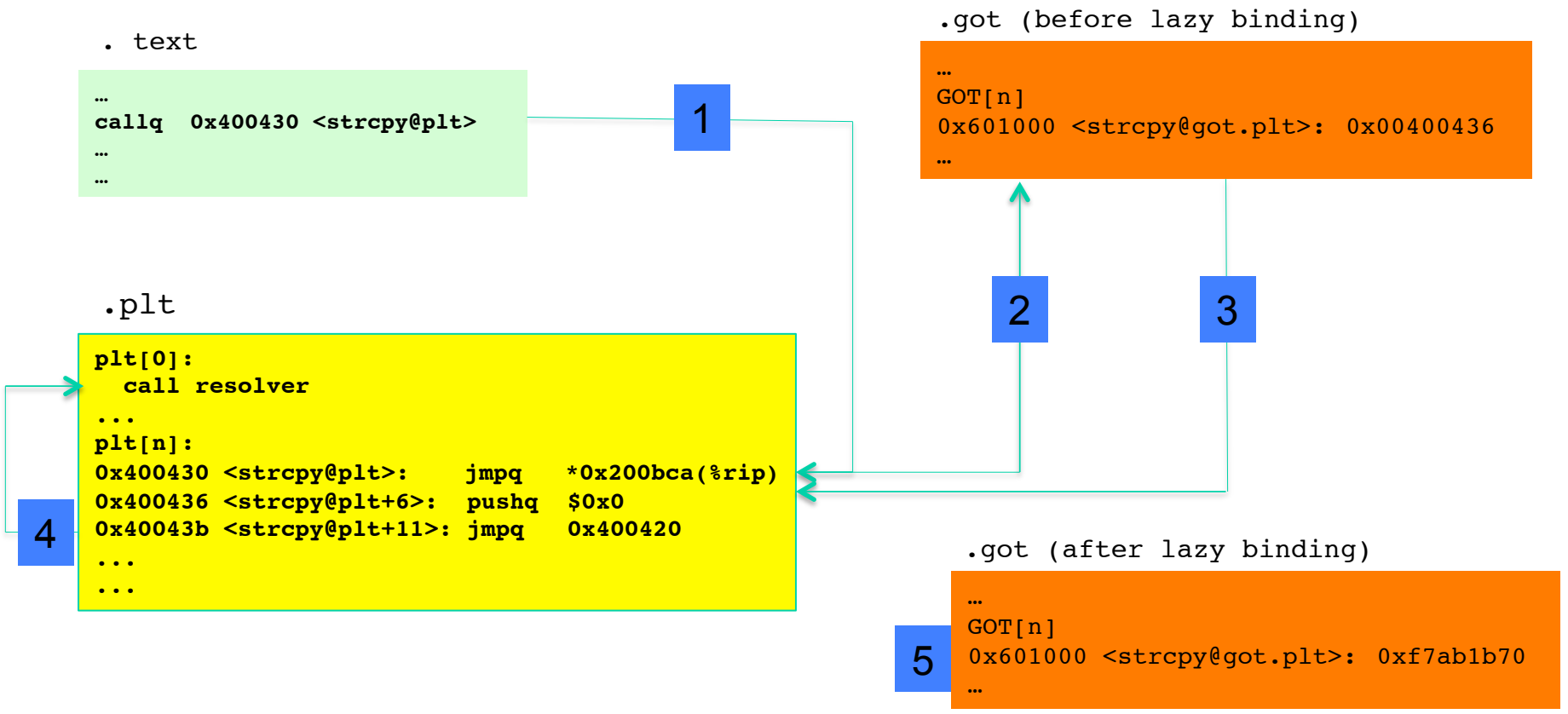
Lez. 7

Hijacking the Global Offset Table (c0ntex)

Procedure Linkage Table (PLT)
Global Offset Table (GOT)

- PLT and GOT, both are sections of an ELF executable file
- Used for the resolution of library functions
- Play a crucial role when lazy binding is adopted

How it works


. text
…
callq 0x400430 <strcpy@plt>
…
…
1
.got (before lazy binding)
…
GOT[n]
0x601000 <strcpy@got.plt>: 0x00400436
…
2
3
.plt
plt[0]:
call resolver
...
plt[n]:
0x400430 <strcpy@plt>: jmpq *0x200bca(%rip)
0x400436 <strcpy@plt+6>: pushq $0x0
0x40043b <strcpy@plt+11>: jmpq 0x400420
...
...
4
.got (after lazy binding)
5
…
GOT[n]
0x601000 <strcpy@got.plt>: 0xf7ab1b70
…

The attack

- The got must be writable at runtime
- If we know where it is we can overwrite its entries with badcode addresses and force the execution of “arbitrary” code
- For doing this the attacked code attack has to provide a pointer variable which we can overwrite (via buffer overflow) with suitable values
- The attack works also with non executable stack and stackguard protections activated

COUNTERMEASURES

```
dbruschi@ubuntu:~$ ./checksec.sh --kernel
* Kernel protection information:

  Description - List the status of kernel protection mechanisms. Rather than
  inspect kernel mechanisms that may aid in the prevention of exploitation of
  userspace processes, this option lists the status of kernel configuration
  options that harden the kernel itself against attack.

  Kernel config: /boot/config-3.5.0-23-generic

  Warning: The config on disk may not represent running kernel config!

  GCC stack protector support:            Enabled
  Strict user copy checks:                Disabled
  Enforce read-only kernel data:          Enabled
  Restrict /dev/mem access:               Enabled
  Restrict /dev/kmem access:              Enabled

* grsecurity / PaX: No GRKERNSEC

  The grsecurity / PaX patchset is available here:
    http://grsecurity.net/

* Kernel Heap Hardening: No KERNHEAP

  The KERNHEAP hardening patchset is available here:
    https://www.subreption.com/kernheap/
```

```
dbruschi@ubuntu:~$ ./checksec.sh --file Didattica/exe/got
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   Didattica/exe/got
dbruschi@ubuntu:~$ ./checksec.sh --file /bin/login
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /bin/login
dbruschi@ubuntu:~$ ./checksec.sh --file /bin/bash
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   /bin/bash
```

Code injection attacks

- Three essential stages:
  1. Inject attack code
  2. Hijack control flow
  3. Execute attacker code

Defenses

- Strategies
  - Detect and remove vulnerabilities (best)
  - Detect and prevent code injection
  - Detect and prevent control flow hijacking
  - Detect and prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

Defenses

- Good Programming Practices
- KERNEL
  - Non-executable parts of the address space
    - Solar Designer's "non-exec stack patch", Exec Shield, OpenBSD's W^X, XP SP2 NX
  - Address space randomization
- Compiler based protector
  - StackGuard, stack shield, propolice, XP SP2 /Gs
- Runtime stack integrity checker
  - Libsafe

✓ There is no single solution!!!

Good programming practices

Prevention

- Don’t use C or C++ (use type-safe language)
  - Legacy code
  - Practical?
- Better programmer awareness & training
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler, www.dwheeler.com/secure-programs
  - Secure C coding, T. Sirainen www.irccrew.org/~cras/security/c-guide.html

Secure Coding

- Avoid risky programming constructs
  - Use fgets instead of gets
  - Use strn* APIs instead of str* APIs
  - Use snprintf instead of sprintf and vsprintf
  - scanf & printf: use format strings
- Never assume anything about inputs
  - Negative value, big value
  - Very long strings

Better string libraries

- **libsafe.h** provides safer, modified versions of string functions eg:
  - **strlcpy**(dst,src,size) and **strlcat**(dst,src,size**)** with the size of dst, not the maximum length copied
  - Used in OpenBSD
- **glib.h** provides Gstring type for dynamically growing null-terminated strings in C
  - but failure to allocate will result in crash that cannot be intercepted, which may not be acceptable
- **Strsafe.h** by Microsoft guarantees null-termination and always takes destination size as argument
- **C++ string class**
  - data() and c-str()return low level C strings, ie char*, with result of data() is not always null-terminated on all platforms...

Bugs to Detect in Source Code Analysis

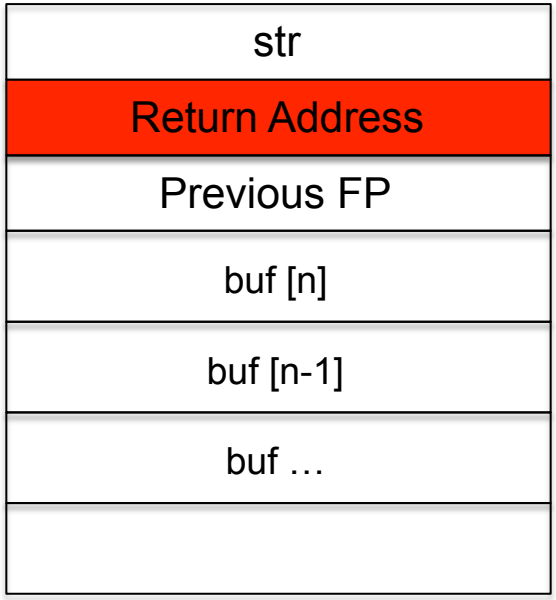
- Some examples

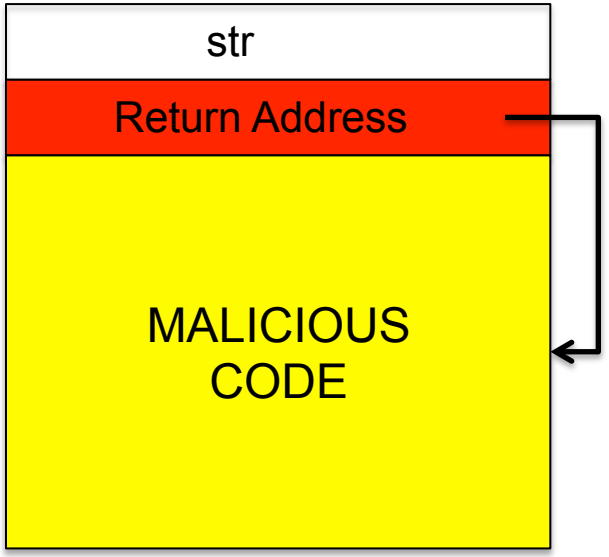
- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Kernel level defenses

Non executable part of address space

Buffer overflows





Why should the machine interpret stack data as instructions?

Noexec: W^X (W xor X)

- The idea: mark memory page as either WRITABLE or EXECUTABLE but not both
- Specifically: mark STACK and HEAP non executable
  - AMD64: NX bit (Non---Executable)
  - IA---64: XD bit (eXecute Disabled)
  - ARMv6: XN bit (eXecute Never)
- Extra bit in each page table entry
- Processor do not execute code if NX bit = 1
- Mark heap and stack segments as such

Noexec: software solutions

- Software emulation of W^X
  - ExecShield (RedHat, Linux)
  - PaX (Page---eXec) (uses NX bit if available)
- PaX: Kernel patch which provides non-executable memory pages and full address space layout randomization (ASLR) for a wide variety of architectures.

Noexec can stop

- Stack Smashing by Alephone
- Stack smash that overwrites pointer to point at shell code in Heap or Env variable
- Some kind of Heap overflows
- IT CAN DO NOTHING against RETURN TO LIBC and GOT OVERWRITING ATTACKS !!!

PAX

PaX

- Linux kernel patch
- Goal: prevent execution of arbitrary code in an existing process's memory space
- Enable executable/non-executable memory pages
- Any section not marked as executable in ELF binary is non-executable by default
  - Stack, heap, anonymous memory regions
- Access control in mmap(), mprotect() prevents unsafe changes to protection state at runtime
- Randomize address space layout

PaX Components

- SEGMEXEC
- PAGEEXEC
- KERNEXEC
- ASLR
  - RANDMMAP
  - RANDEXEC
  - ET_DYN
  - RANDKSTACK

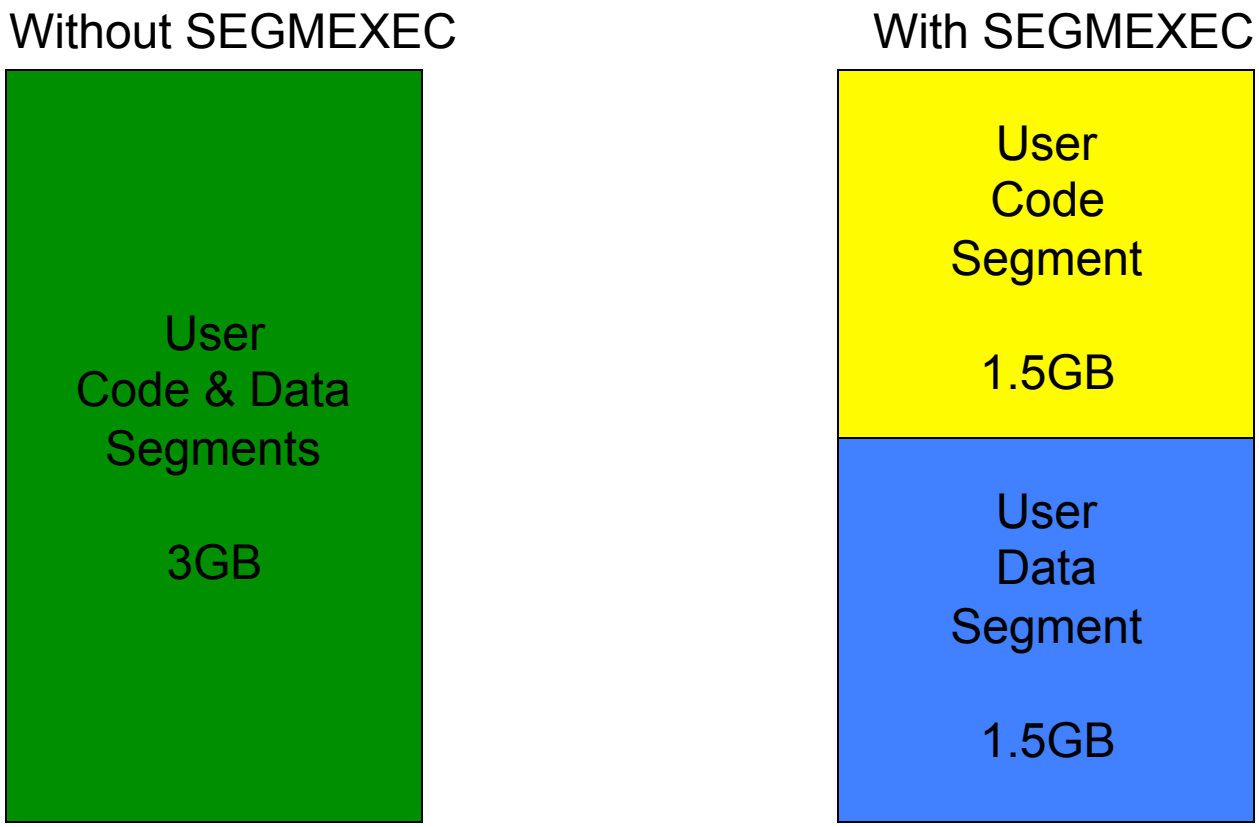
PaX - SEGMEXEC

- SEGMEXEC is PaX's implementation of per-page non-executable user pages using the segmentation logic of IA-32 (Intel x86 architecture) and virtual memory area mirroring (developed by PaX).

PaX SEGMEXEC

- Derived from the IA-32 processor segmentation logic
- Linux runs in protected mode with paging enabled on IA-32 processors, which means that each address translation requires a two step process.
  - LOGICAL <-> LINEAR <-> PHYSICAL
- The 3Gb of userland memory space is divided in half:
  - Data Segment: 0x00000000 - 0x5fffffff
  - Code Segment: 0x60000000 – 0xbfffffff
- Page fault is generated if instruction fetches are initiated in the non-executable pages

PaX – SEGMEXEC (cont.)



PaX – SEGMEXEC (cont.)

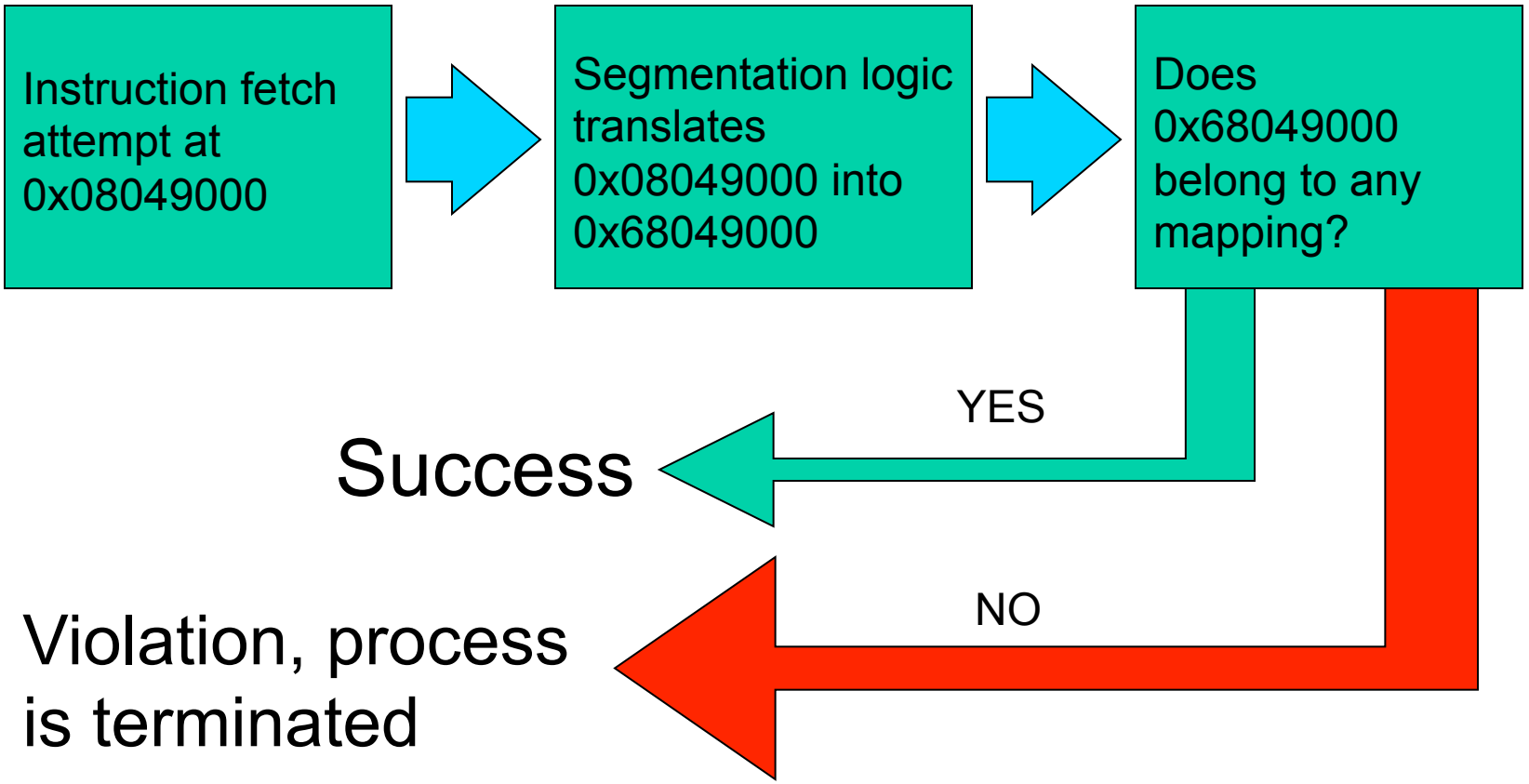
- Since executable mappings may contain data as well (constant strings, function pointer tables, etc), we have to have a mirror of these mappings at the same logical addresses in the data segment as well (0-1.5 GB range in linear address space)
- The nice property of this setup is that a pair of mirrored regions will have a constant difference between their start/end addresses: 1.5 GB (or SEGMEXEC_TASK_SIZE as it is often referenced in the code

Example

```
Active PaX features: SEGMEXEC and MPROTECT

 [1] 08048000-0804a000 R-Xp 00000000 00:0b 1109       /tmp/cat
 [2] 0804a000-0804b000 RW-p 00002000 00:0b 1109       /tmp/cat
 [3] 0804b000-0804d000 RW-p 00000000 00:00 0
 [4] 20000000-20015000 R-Xp 00000000 03:07 110818     /lib/ld-2.2.5.so
 [5] 20015000-20016000 RW-p 00014000 03:07 110818     /lib/ld-2.2.5.so
 [6] 2001e000-20143000 R-Xp 00000000 03:07 106687     /lib/libc-2.2.5.so
 [7] 20143000-20149000 RW-p 00125000 03:07 106687     /lib/libc-2.2.5.so
 [8] 20149000-2014d000 RW-p 00000000 00:00 0
 [9] 5fffe000-60000000 RW-p fffff000 00:00 0
[10] 68048000-6804a000 R-Xp 00000000 00:0b 1109       /tmp/cat
[11] 80000000-80015000 R-Xp 00000000 03:07 110818     /lib/ld-2.2.5.so
[12] 8001e000-80143000 R-Xp 00000000 03:07 106687     /lib/libc-2.2.5.so
```

PaX – SEGMEXEC (cont.)



PaX - PAGEEXEC

- PAGEEXEC was PaX's first implementation of non-executable pages.
- Because of SEGMEXEC, it's not used anymore on x86
- Platforms which support the executable bit in hardware are implemented under PAGEEXEC (currently alpha, ppc, parisc, sparc, sparc64, amd64, and ia64)

mprotect() in PaX

- mprotect() is a Linux kernel routine for specifying desired protections for memory pages
- PaX modifies mprotect() to prevent:
  - Creation of executable anonymous memory mappings
  - Creation of executable and writable file mappings
  - Making executable, read-only file mapping writable
    - Except when relocating the binary
  - Conversion of non-executable mapping to executable

Access Control in PaX mprotect()

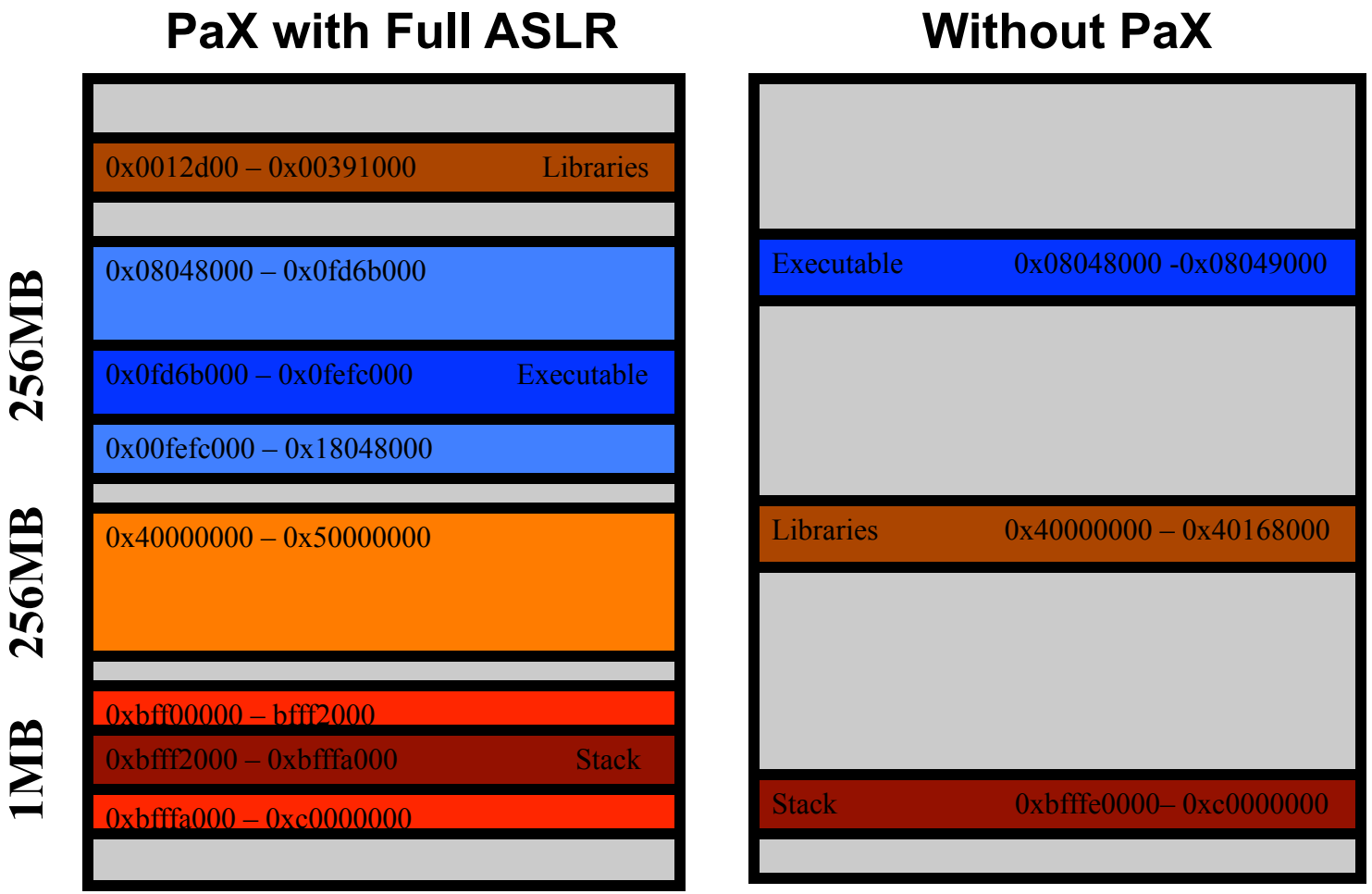
- In standard Linux kernel, each memory mapping is associated with permission bits
  - VM_WRITE, VM_EXEC, VM_MAYWRITE, VM_MAYEXEC
    - Stored in the vm_flags field of the vma kernel data structure
    - 16 possible write/execute states for each memory page
- PaX makes sure that the same page cannot be writable AND executable at the same time
  - Ensures that the page is in one of the 4 “good” states
    - VM_MAYWRITE, VM_MAYEXEC, VM_WRITE | VM_MAYWRITE, VM_EXEC | VM_MAYEXEC
  - Also need to ensure that attacker cannot make a region executable when mapping it using mmap()

RANDOMIZATION

Randomization: Motivations.

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Prevention in grsecurity - PaX



Address Space Layout Randomization

- Arranging the positions of key data areas randomly in a process' address space.
  - e.g., the base of the executable and position of libraries (libc), heap, and stack,
  - Effects: for return to libc, needs to know address of the key functions.
  - Attacks:
    - Repetitively guess randomized address
    - Spraying injected attack code
- Vista has this enabled, software packages available for Linux and other UNIX variants

PaX ASLR

- Address Space Layout Randomization (ASLR) renders exploits which depend on predetermined memory addresses useless by randomizing the layout of the virtual memory address space.

- PaX implementation of ASLR consists of:
  - RANDUSTACK
  - RANDKSTACK
  - RANDMMAP
  - RANDEXEC

PaX RANDUSTACK

- Responsible for randomizing userspace stack
- Userspace stack is created by the kernel upon each execve() system call
  - Allocates appropriate number of pages
  - Maps pages to process's virtual address space
    - Userspace stack is usually mapped at 0xBFFFFFFF, but PaX chooses a random base address
- In addition to base address, PaX randomizes the range of allocated memory

PaX RANDKSTACK

- Each task is assigned two pages of kernel memory to be used during the execution of system calls, interrupts, and exceptions.
- Each system call is protected because the kernel stack pointer will be at the point of initial entry when the kernel returns to userspace
- This behaviour means that a userland originating attack against a kernel bug would find itself always at the same place on the task's kernel stack

PaX RANDMMAP

- Linux usually allocates heap space by beginning at the base of a task's unmapped memory and locating the nearest chunk of unallocated space which is large enough.

- RANDMMAP modifies this functionality in do_mmap() by adding a random delta_mmap value to the base address before searching for free memory.

PaX RANDEXEC

- Randomizes location of ELF binaries in memory
- Problem if the binary was created by a linker which assumed that it will be loaded at a fixed address and omitted relocation information
  - PaX maps the binary to its normal location, but makes it non-executable + creates an executable mirror copy at a random location
  - Access to the normal location produces a page fault
  - Page handler redirects to the mirror "if safe"
    - Looks for "signatures" of return-to-libc attacks and may result in false positives

PaX – ASLR (cont.)

- Notes on amount of randomization:
  - The following values are for 32bit architectures. They are larger on 64bit architectures, though not twice as large (since they generally don't use 64 bits for the address space).
    - Stack – 24 bits (28 bits for argument/environment pages)
    - Mmap – 16 bits
    - Executable – 16 bits
    - Heap – 12 bits (or 24 bits if executable is randomized also)

PaX – ASLR (cont.)

- The randomizations applied to each memory region are independent of each other
  - Because PaX guarantees no arbitrary code execution, exploits will most likely need to access different memory regions.
  - So, if the exploit needs access to libraries and the stack, the bits that must be guessed are the sum of the two regions: 40 bits (or 44). The chance of such an attack succeeding while depending on hard coded addresses is effectively zero.

PaX – ASLR (cont.)

```
08048000-0804c000 r-xp /home/spender/cat
0804c000-0804d000 rw-p /home/spender/cat
0804d000-08078000 rw-p
4edaa000-4edbe000 r-xp /lib/ld-2.3.2.so
4edbe000-4edbf000 rw-p /lib/ld-2.3.2.so
4edbf000-4edc0000 rw-p
4edc8000-4eeef000 r-xp /lib/libc-2.3.2.so
4eeef000-4eef4000 rw-p /lib/libc-2.3.2.so
4eef4000-4eef6000 rw-p
4eef6000-4f07b000 r--p /usr/lib/locale/locale-archive
bf3dc000-bf3dd000 rw-p
```

```
08048000-0804c000 r-xp /home/spender/cat
0804c000-0804d000 rw-p /home/spender/cat
0804d000-08070000 rw-p
43d8c000-43da0000 r-xp /lib/ld-2.3.2.so
43da0000-43da1000 rw-p /lib/ld-2.3.2.so
43da1000-43da2000 rw-p
43daa000-43ed1000 r-xp /lib/libc-2.3.2.so
43ed1000-43ed6000 rw-p /lib/libc-2.3.2.so
43ed6000-43ed8000 rw-p
43ed8000-4405d000 r--p /usr/lib/locale/locale-
archive
b54f9000-b54fa000 rw-p
```

Two runs of a binary with stack, mmap, and heap randomization

PaX – ASLR (cont.)

- ET_DYN
  - Special type of ELF binary (the same used for shared libraries)
  - Position independent code (PIC)
  - Allows for relocation of the binary at a random location
  - Needed to achieve Full ASLR
  - Requires a recompile and re-link of applications
  - Adamantix and Hardened Gentoo have adopted these changes.

Instruction Set Randomization

- Instruction Set Randomization (ISR)
  - Each program has a *different* and *secret* instruction set
  - Use translator to randomize instructions at load-time
  - Attacker cannot execute its own code.
- What constitutes instruction set depends on the environment.
  - for binary code, it is CPU instruction
  - for interpreted program, it depends on the interpreter

Instruction Set Randomization

- An implementation for x86 using the Bochs emulator
  - network intensive applications doesn't have too much performance overhead
  - CPU intensive applications have one to two orders of slow-down
- Not yet used in practice

Compiler based protectors

Stackguard

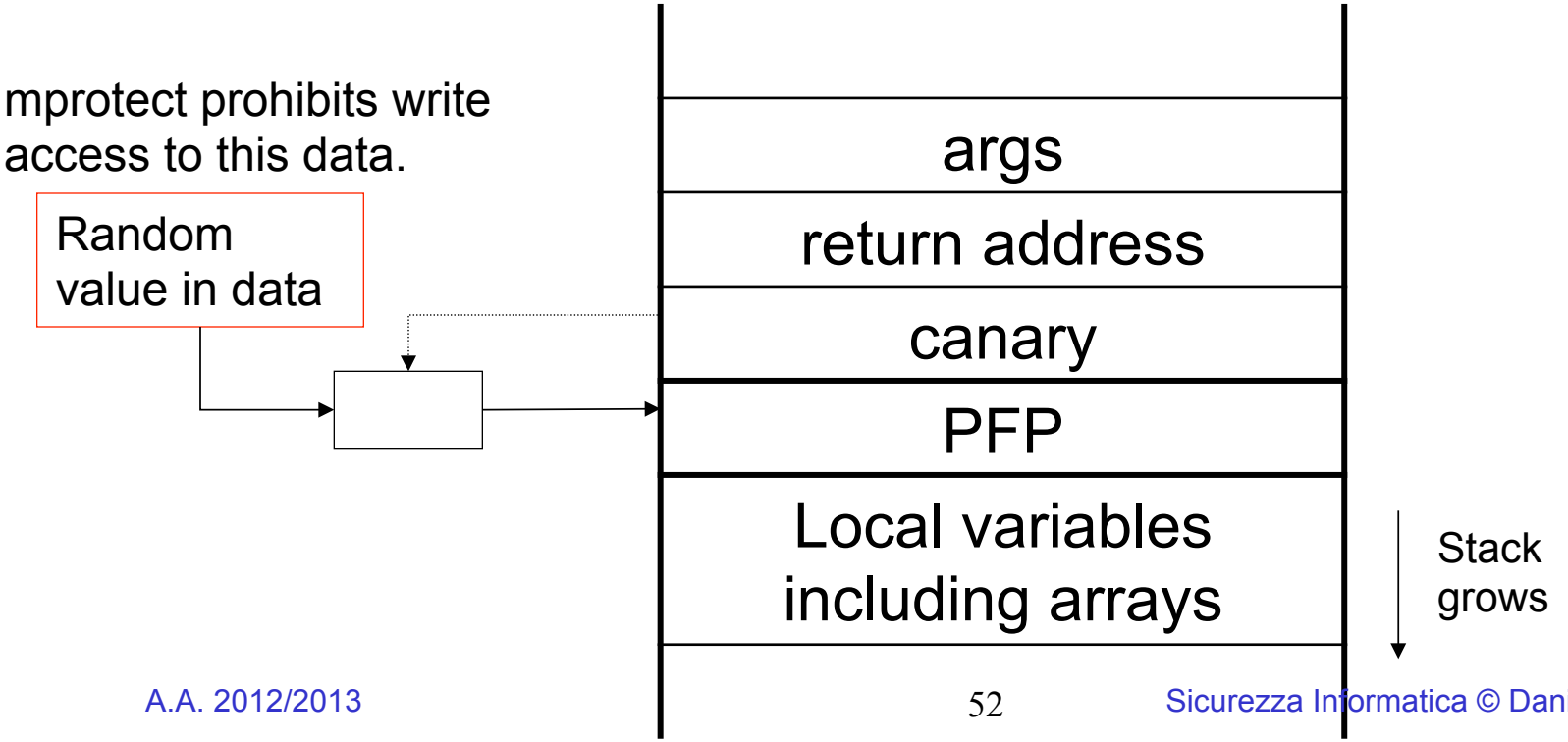
- The StackGuard compiler invented and implemented by Crispin Cowan et al. is perhaps the most well referenced of the current dynamic intrusion prevention techniques
- It is designed for detecting and stopping stackbased buffer overflows targeting the return address
- Currently embedded in the GCC compiler

The key idea

- In stack buffer overflows the attacker has to overwrite local variables, overwrite the old base pointer until it finally reaches the return address
- If we place a dummy value in between the return address and the stack data above, and then check whether this value has been overwritten or not before we allow the return address to be used
- We could detect this kind of attack and possibly prevent it

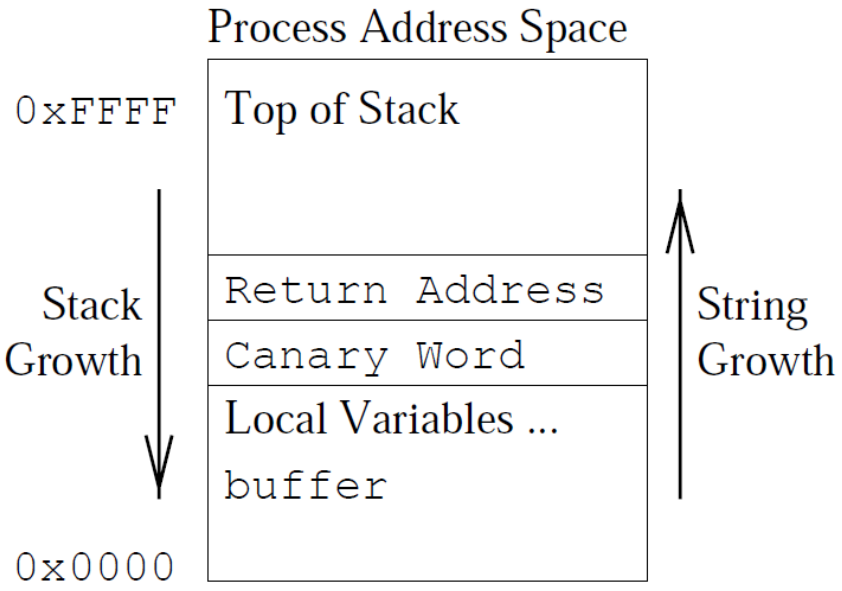
Stack Guard 2.1

- Canary value variations
  - Terminator canary 0x000aff0d
  - Random canary random
  - XOR canary: random ^ return addres



StackGuard, 1998

- Compile in integrity checks for activation records
  - Insert a “canary word”
- If the canary word is damaged, then your stack is corrupted
  - Instead of jumping to attacker code, abort the program
  - Log the intrusion attempt



How does it work

- StackGuard's prologue is different. The first thing it does is pushing a canary into the stack (for StackGuard v2.0.1 it's a constant 0x000aff0d , latter we'll see why), then it continues with standard prologue

```
function_prologue:
    pushl   $0x000aff0d     // push canary into the stack
    pushl   %ebp            // save frame pointer
    mov     %esp,%ebp       // saves a copy of current %esp
    subl    $108, %esp      // space for local variables

    (function body)
```

How does it work

- On the epilogue, StackGuard checks the stack to see if the canary is still there unchanged, if so, it keeps going with normal execution flow, if not it aborts the program

```
function_epilogue:
    leave                           // standard epilogue
    cmpl   $0x000aff0d,(%esp)       // check canary
    jne    canary_changed
    addl   $4,%esp                  // remove canary from stack
    ret

canary_changed:
    ...                             // abort the program with error
    call   __canary_death_handler
    jmp    .                        // just in case I guess
```

How does it work

No stackguard

↑ lower addresses

func()	var2	4 bytes
func()	buf	80 bytes
func()	var1	4 bytes
func()	saved %ebp	4 bytes
func()	return address	4 bytes
main()/func()	func()'s arguments	4 bytes
main()	p	4 bytes
main()	saved %ebp	4 bytes
main()	return address	4 bytes
start()/main()	main()'s arguments	12 bytes

↓ higher addresses

Stackguard

↑ lower addresses

func()	var2	4 bytes
func()	buf	80 bytes
func()	var1	4 bytes
func()	saved %ebp	4 bytes
func()	canary (0x000aff0d)	4 bytes
func()	return address	4 bytes
main()/func()	func()'s arguments	4 bytes
main()	p	4 bytes
main()	saved %ebp	4 bytes
main()	canary (0x000aff0d)	4 bytes
main()	return address	4 bytes
start()/main()	main()'s arguments	4 bytes

↓ higher addresses

How does it work

- If we try to write 0x000aff0d over the former canary (effectively not changing it), the 0x00 will stop strcpy () , and we won't be able to alter the return address . If gets() were used instead of strcpy() to read into buf, we would be able to write 0x00 but 0x0a would stop it
- This type of canaries is called terminator canaries, and is the only canary type StackGuard 2.0.1 can use

Stack Guard

- That's how StackGuard's protection works, and is where the three different flaws can be understood, using STACKGUARD we can still alter
  - local variables located after buf (var1) as they are not protected at all,
  - the saved frame pointer
  - function's arguments, as the check would only be detected after the function finishes, giving the attacker a code window to play with

StackGuard Prototype

- Written in a few days by one intern
- Less than 100 lines of code patch to GCC
  - Helped a lot that the GCC function preamble and function post amble code generator routines were nicely isolated
- First canary was hardcoded 0xDEADBEEF
  - Easily spoofable, but worked for proof of concept

StackGuard

- Microsoft Visual Studio: /gs
  - Uses exactly the StackGuard defense
  - Introduced in 2003; people who were there say that it was independently innovated
- Even though introduced 5 years after StackGuard, Microsoft beat the Linux/FOSS community into mainstream adoption by several years

Canary Spoof Resistance

- The random canary:
  - Pull a random integer from the OS /dev/random at process startup time
  - Simple in concept, but in practice it is very painful to make reading from /dev/random work while still inside crt0.o
  - Made it work, but motivated us to seek something simpler
- "Terminator" canary:
  - CR, LF, 00, -1: the symbols that terminate various string library functions
  - Rationale: will cause all the standard string mashers to terminate while trying to write the canary → cannot spoof the canary and successfully write beyond it
  - Still vulnerable to attacks against poorly used memcpy() code, but buffer overflows thought to be rare

XOR Random Canary

- 1999, “Emsi” creates the frame pointer attack
  - Frame pointer stored below the canary → corruptible
  - Change FP to point to a *fake* activation record constructed on the heap
  - Function return code will believe FP, interpret the fake activation record, and jump to shell code
  - Bypasses both Terminator and Random Canaries
- XOR Random Canary
  - XOR the correct return address with the random canary
  - Integrity check must match both the random number, and the correct return address

Ubuntu

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005f4 <+0>:     push   %rbp
   0x00000000004005f5 <+1>:     mov    %rsp,%rbp
   0x00000000004005f8 <+4>:     sub    $0x70,%rsp
   0x00000000004005fc <+8>:     mov    %fs:0x28,%rax
   0x0000000000400605 <+17>:    mov    %rax,-0x8(%rbp)
   0x0000000000400609 <+21>:    xor    %eax,%eax
   0x000000000040060b <+23>:    mov    $0x40074c,%eax
   0x0000000000400610 <+28>:    lea    -0x64(%rbp),%rdx
   0x0000000000400614 <+32>:    lea    -0x60(%rbp),%rcx
   0x0000000000400618 <+36>:    mov    %rcx,%rsi
   0x000000000040061b <+39>:    mov    %rax,%rdi
   0x000000000040061e <+42>:    mov    $0x0,%eax
   0x0000000000400623 <+47>:    callq  0x4004e0 <printf@plt>
   0x0000000000400628 <+52>:    lea    -0x60(%rbp),%rax
   0x000000000040062c <+56>:    mov    %rax,%rdi
   0x000000000040062f <+59>:    callq  0x400500 <gets@plt>
   0x0000000000400634 <+64>:    mov    -0x64(%rbp),%eax
   0x0000000000400637 <+67>:    cmp    $0xd0a00,%eax
   0x000000000040063c <+72>:    jne    0x400648 <main+84>
   0x000000000040063e <+74>:    mov    $0x400765,%edi
   0x0000000000400643 <+79>:    callq  0x4004c0 <puts@plt>
   0x0000000000400648 <+84>:    mov    -0x8(%rbp),%rdx
   0x000000000040064c <+88>:    xor    %fs:0x28,%rdx
   0x0000000000400655 <+97>:    je     0x40065c <main+104>
   0x0000000000400657 <+99>:    callq  0x4004d0 <__stack_chk_fail@plt>
   0x000000000040065c <+104>:   leaveq
   0x000000000040065d <+105>:   retq
End of assembler dump.
```

Other Stack Smashing Defenses

- StackShield:
  - Copied valid return addresses to safe memory, check them on function return
  - Implemented as a modified assembler → requires hacking your makefiles
- Libsafe: armored variants of the “big 7” standard string library functions
  - Library code does a plausability check on the parameters; ensure that they are not pointing back up the stack at an activation record
  - Advantage: no recompile necessary
  - Disadvantage: no protection for hand-coded string handling, or anything other than the big-7

Other Stack Smashing Defenses

- StackGhost: uses SPARC CPU hardware to get OS in the loop to armor the stack
- Hardware: numerous papers proposing “slightly” modified CPU hardware to protect against stack smashing
  - Typically protection about as good as StackGuard
  - Advantage: don’t have to re-compile code
  - Disadvantage: do have to re-compile code to run on non-existent hardware, which tends to limit adoption ☺

StackGuard Derivatives: ProPolice

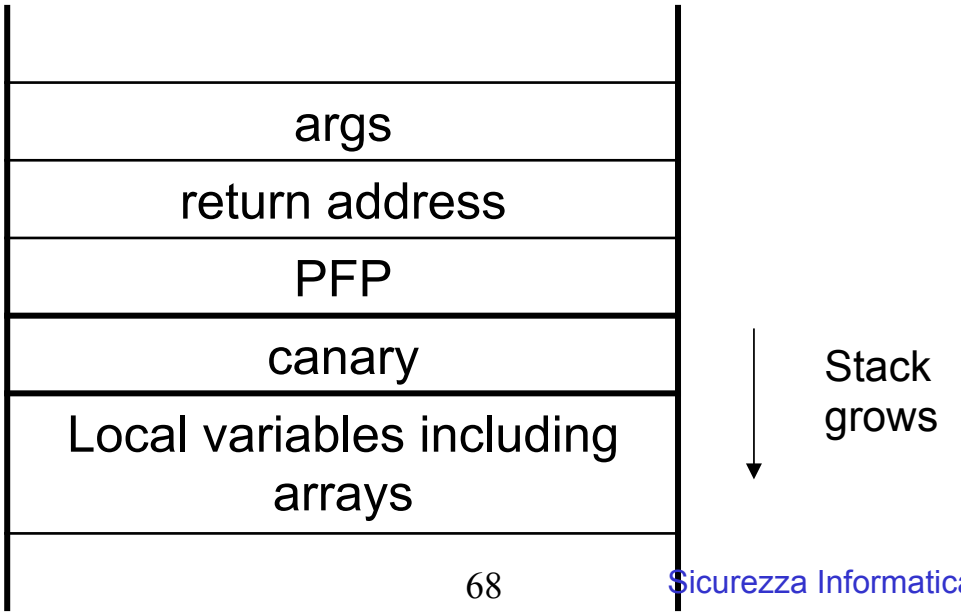
- IBM Research Japan
  - Also a modified GCC
  - Copied StackGuard defense exactly, and acknowledged it
  - Enhanced with variable sorting: sort buffers (arrays) up to the top of local variables, so that they cannot overflow other important values
- Used a different code generator technique
  - More compatible with the newer code generator architecture in GCC 2 and GCC 3
  - Ultimately ProPolice is what is adopted into GCC and became the `-fstack_protector` feature

Stack Smashing Protector (SSP)/ Propolice: design goal

- Introduce "Safe Stack Usage Model"
  - This is a combination of an ideal stack layout and a way to check the stack integrity
- Transform a program to meet the ideal stack layout as much as possible:
  - Makes use of canary values by rearranging local variables and function pointers
  - A patch for GNU gcc compiler adds a compilation stage to transform the program
- It is developed by Hiroaki Etoh at IBM

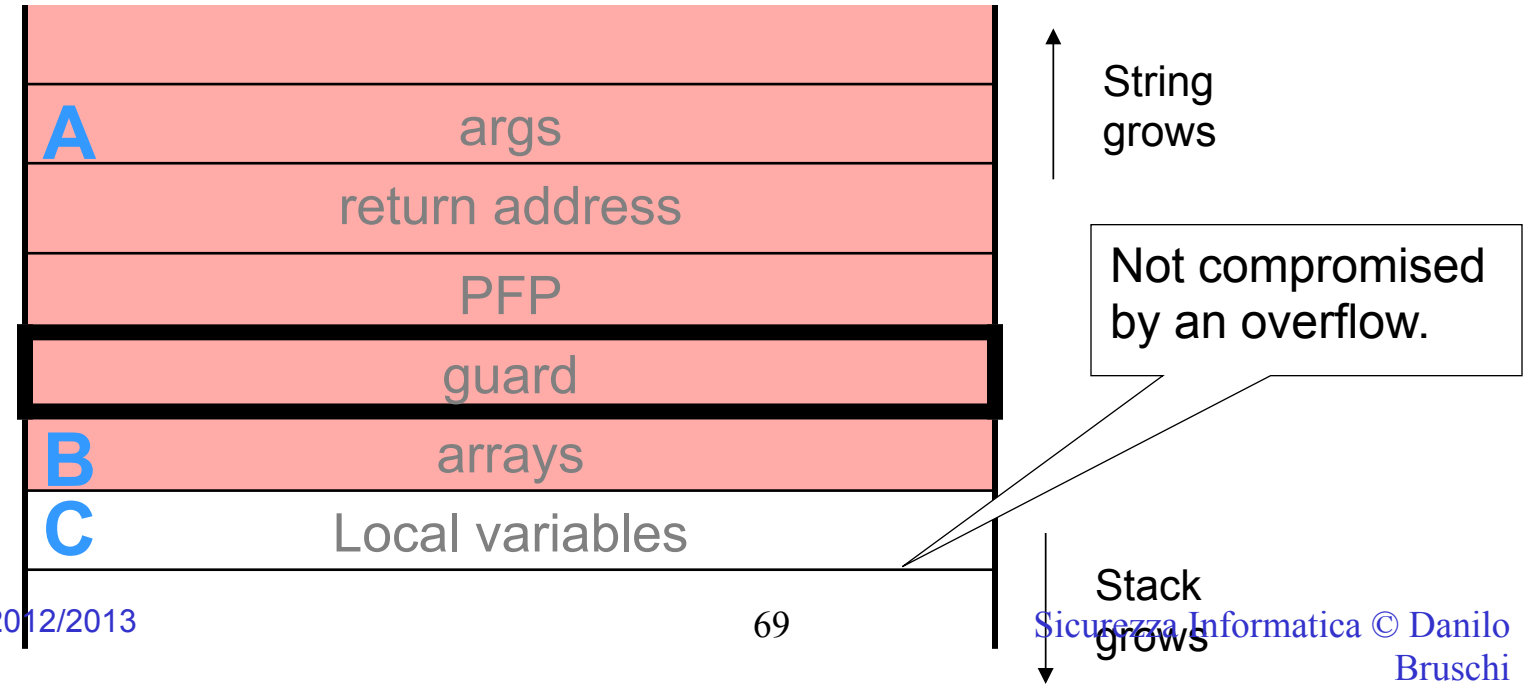
Stack integrity

- Stack integrity check:
  - Move the canary to eliminate the frame pointer problem
  - Assigns unpredictable value into the guard at the function prologue. Confirms the integrity of the guard value at the function epilogue, or aborts the program execution



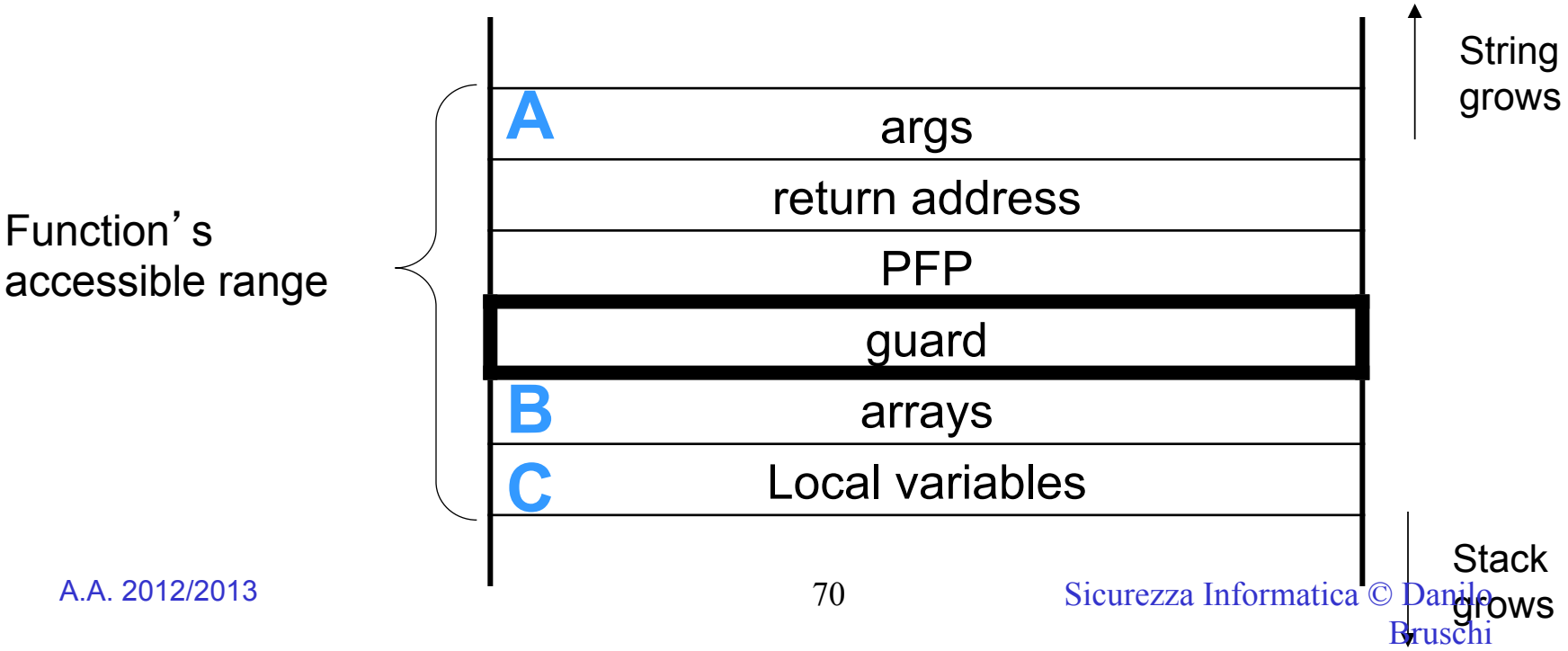
Safe Stack Usage Model

- Ideal stack layout:
  - **A** doesn't have arrays nor pointer variables.
  - **B** has only arrays
  - **C** has no array, but has pointer variables.



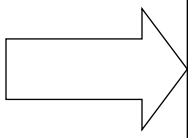
Why caller function is safe from a stack smashing attack

- There are no pointer variables from args to guard, which is the function's accessible range. So any memory can't be compromised by a pointer attack.
- When a function successfully return to the caller function, it means that contiguous chunk of memory of caller function's stack is not compromised by buffer overflows.



Instrument PFP and arrays: intuitive explanation

```
foo () {
    char *p;
    char buf[128];
    gets (buf);
}
```

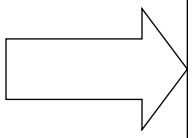


```
int32  random_number;
foo () {
    volatile int32 guard;
    char buf[128];
    char *p;
    guard = random_number;
    gets (buf);
    if (guard != random_number)
/* program halts */
}
```

1. Insert guard instrument
2. Relocate local variables

Intuitive explanation: how to treat function arguments if any of them has a pointer type.

```
foo (int a, void (*fn)()) {
    char buf[128];
    gets (buf);
    (*fn)();
}
```



```
Int32  random_number;
foo (int a, void (*fn)()) {
    volatile int32 guard;
    char buf[128];
    (void *safefn)() = fn;
    guard = random_number;
    gets (buf);
    (*safefn)();
    if (guard != random_number)
/* program halts */
}
```

1. Copy the pointer argument to a new variable which will be assigned to region **C**.
2. Rename the function call with the assigned variable.

propolice status

http://www.research.ibm.com/trl/projects/security/ssp/

- Actual usage
  - Laser5, trusted debian, openbsd, gentoo, etc
- Supported architectures
  - Ix86, powerpc, alpha, sparc, mips, vax, m68k, amd64
- Gcc versions
  - Gcc 2.95.3 – gcc3.4.1

propolice: stack protector options

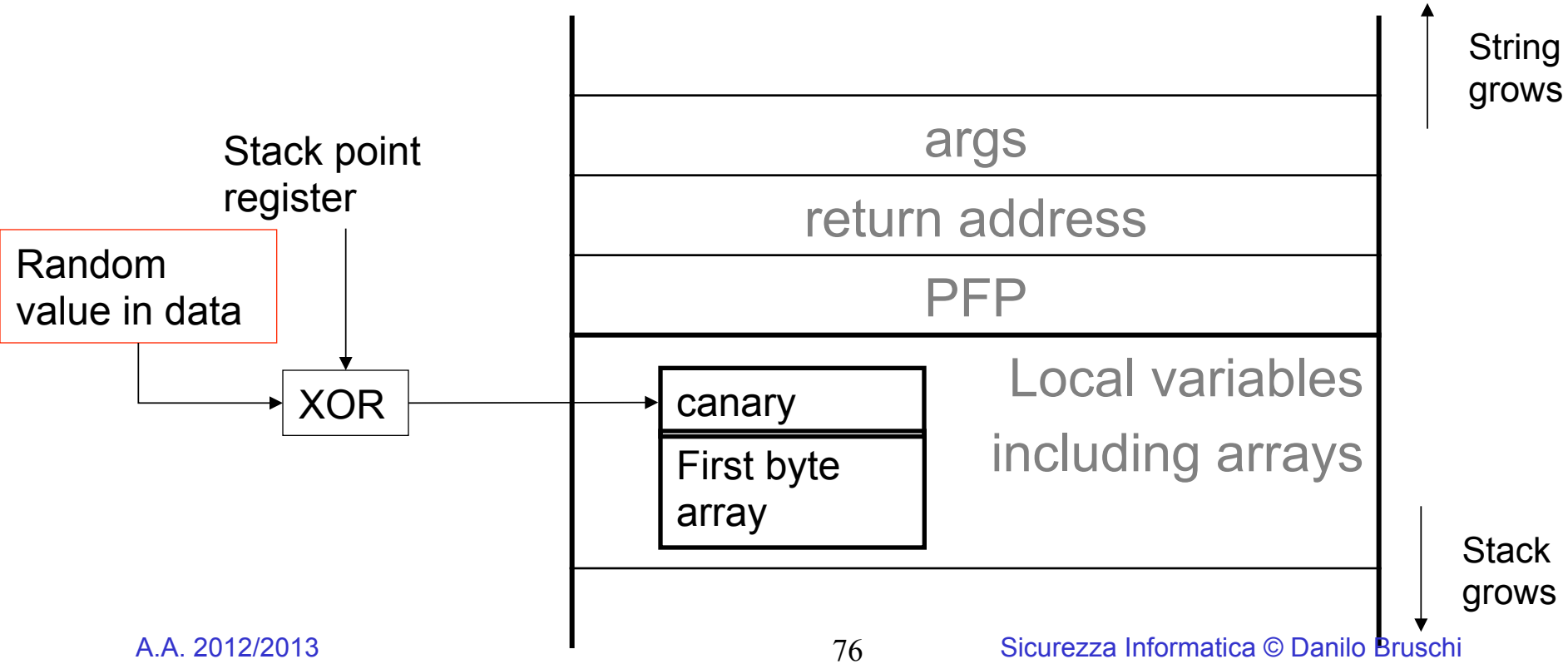
- -fstack-protector
  - Stack protection instruments are generated only when the function has a byte array
- -fstack-protector-all
  - Always generate the guard instrument
  - If a byte array is used, it is allocated next to the guard
  - Otherwise, any array is allocated next to the guard

Microsoft XP SP2
Windows 2003 stack protection

- Non executable stack
- Compiler /Gs option
  - Combination method of xor canary and propolice
  - Far from ideal stack layout
- Vulnerability report
  - David Litchfield, “Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 server”

How Gs option works

- Canary is inserted prior to the first occurrence of byte array allocated
- Local variables except arrays seems to be assigned alphabetical order in the stack.



Conclusion

- None of these protections are perfect!
  - even if attacks to return addresses are caught, integrity of other data other than the stack can still be abused
  - clever attacks may leave canaries intact
  - where do you store the "master" canary value
    - a cleverer attack could change it
  - none of this protects against heap overflows
    - eg buffer overflow within a struct...
  - New proposed non-control attack