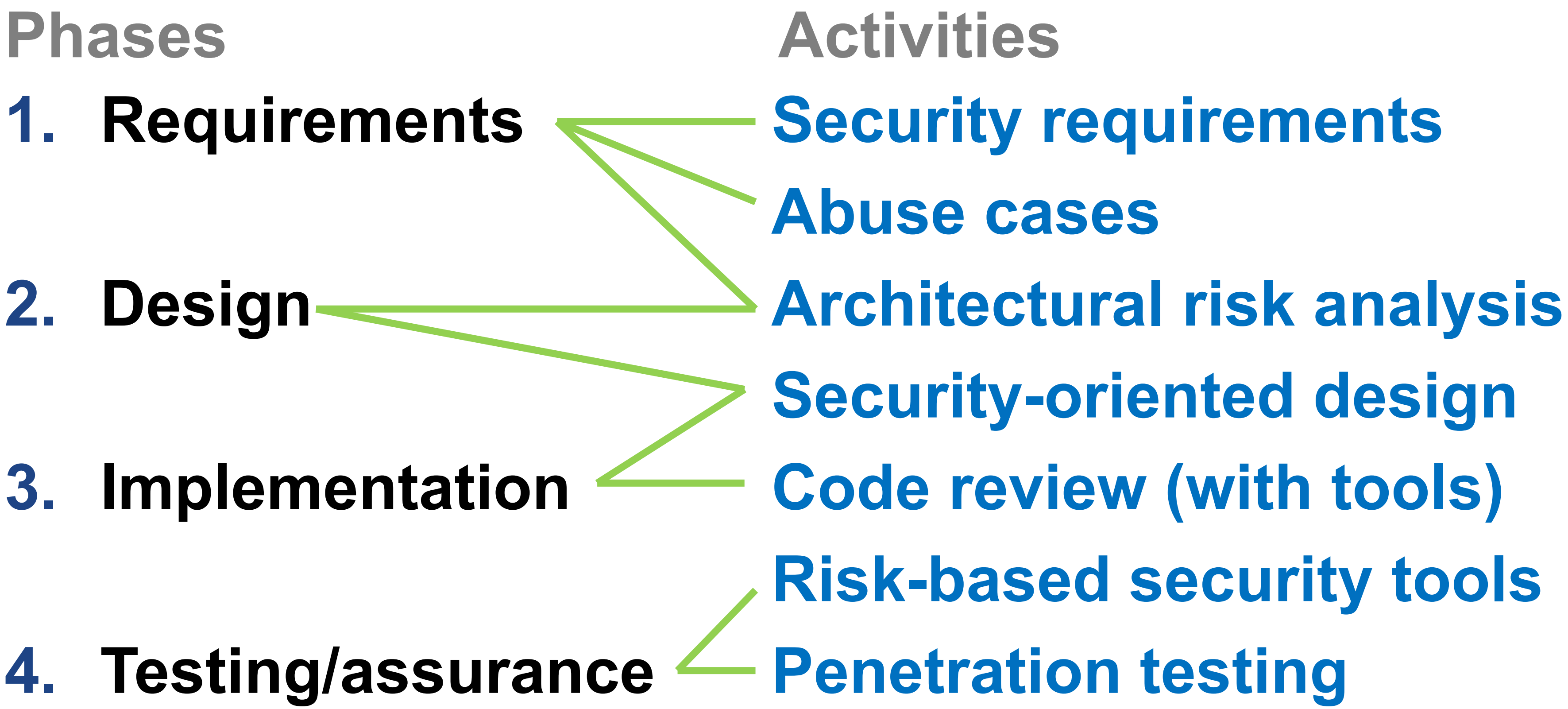
# *PA193 - Secure coding principles and practices*

## Language level vulnerabilities:
## Buffer overflow, type overflow, strings

Petr Švenda svenda@fi.muni.cz

CRoCS
Centre for Research on
Cryptography and Security

# Security engineering – big picture



Phases

1. Requirements
2. Design
3. Implementation
4. Testing/assurance

Activities

- Security requirements
- Abuse cases
- Architectural risk analysis
- Security-oriented design
- Code review (with tools)
- Risk-based security tools
- Penetration testing

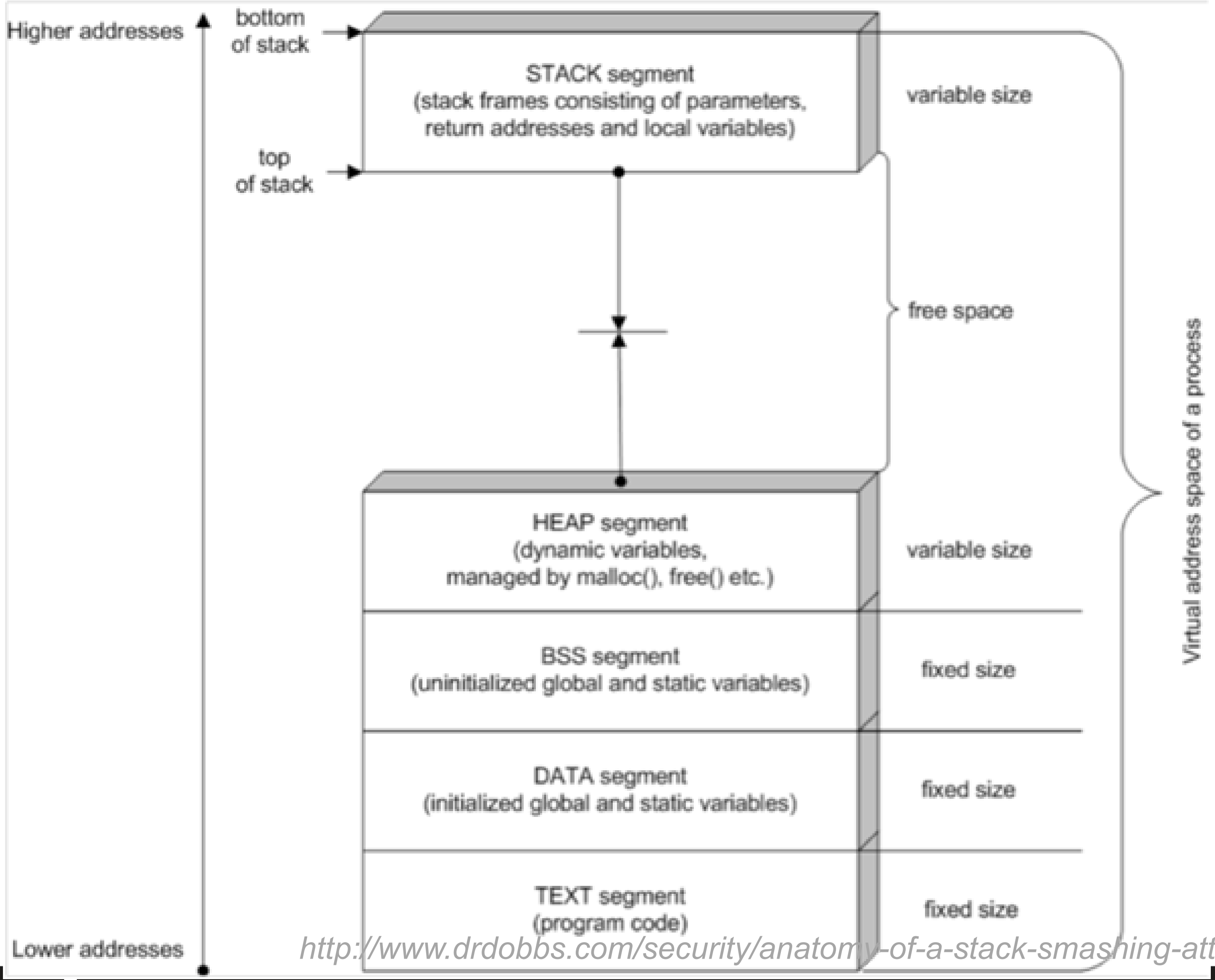
*Based on Software security by M. Hicks (Coursera): https://class.coursera.org/softwaresec-002*

# Overview

- Lecture: problems, prevention
  - buffer overflow (stack/heap/type)
  - string formatting problems
  - compiler protection
  - platform protections (DEP, ASLR)
- Labs
  - compiler flags, buffer overflow exercises

# PROBLEM?

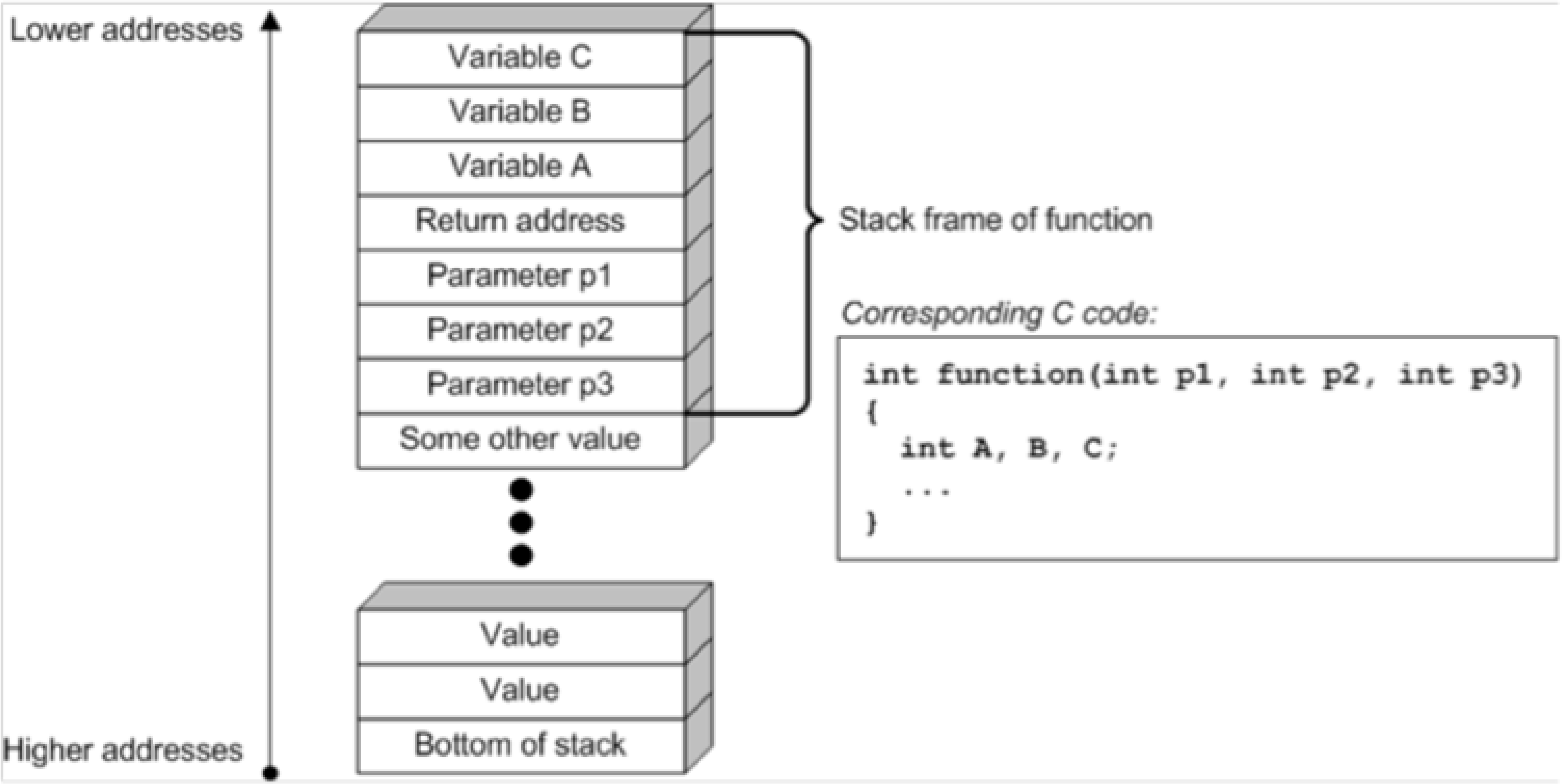
# Process memory layout



Higher addresses
bottom of stack
STACK segment
(stack frames consisting of parameters, return addresses and local variables)
variable size
top of stack
free space
HEAP segment
(dynamic variables, managed by malloc(), free() etc.)
variable size
BSS segment
(uninitialized global and static variables)
fixed size
DATA segment
(initialized global and static variables)
fixed size
TEXT segment
(program code)
fixed size
Lower addresses
Virtual address space of a process

http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

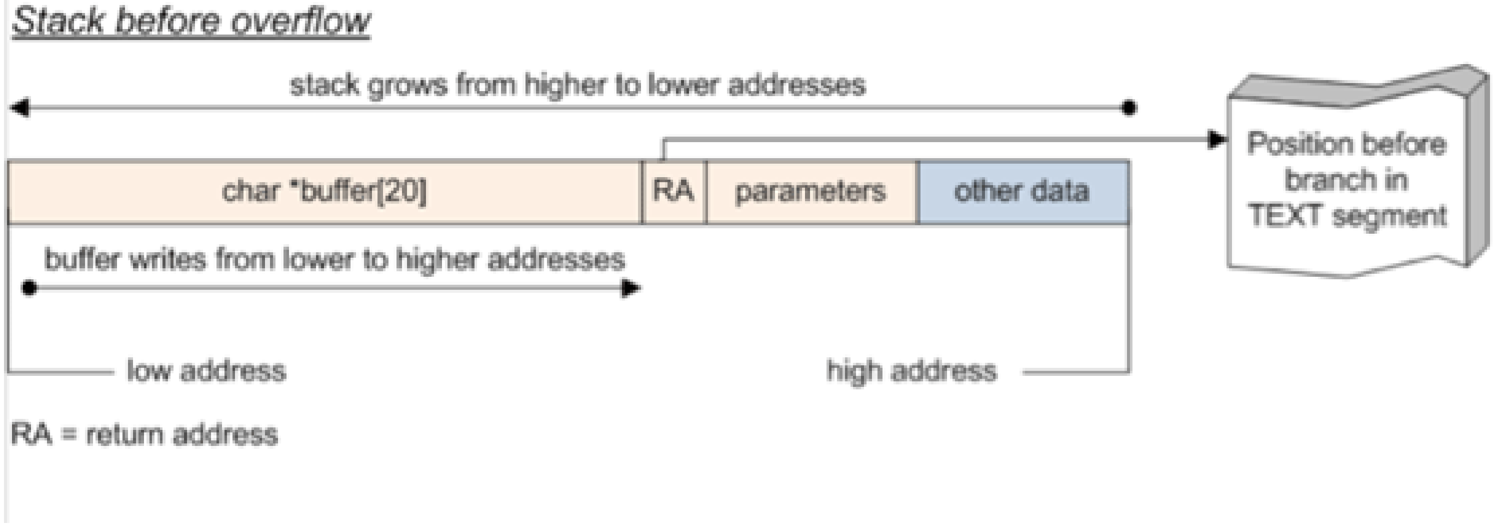
# Stack memory layout



```
int function(int p1, int p2, int p3)
{
  int A, B, C;
   ...
}
```

http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

# Stack overflow


Stack before overflow

stack grows from higher to lower addresses
char *buffer[20]
RA
parameters
other data
Position before branch in TEXT segment
buffer writes from lower to higher addresses
low address
high address

RA = return address

RA = return address

*http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#*

# Memory overflow - taxonomy

1. Buffer overflows
2. Stack overflows
3. Format strings
4. Heap overflows
5. .data/.bss segment overflows

# Buffer overflow vulnerabilities - motivation

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
#define USER_INPUT_MAX_LENGTH  8
char userName[USER_INPUT_MAX_LENGTH];
gets(userName);
```

- Classic buffer overflow as just explained
- Detailed exploitation demo during labs this week

# Type-overflow vulnerabilities - motivation

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
for (unsigned char i = 10; i >= 0; i--) {
    /* ... */
}
```

- And what about following variant?
  - Be aware: char can be both signed (x64) or unsigned (ARM)

```
for (char i = 10; i >= 0; i--) {
    /* ... */
}
```

# Type overflow – basic problem

- Types are having limited range for the values
  - char: 256 values, int: $2^{32}$ values
  - add, multiplication can reach lower/upper limit
  - `char value = 250 + 10` == ?
- Signed vs. unsigned types
  - `for (unsigned char i = 10; i >= 0; i--) {/* ... */ }`
- Type value will underflow/overflow
  - CPU overflow flag is set
  - but without active checking not detected in program

Try this at home!

# Type overflow – example with dynalloc

```
typedef struct _some_structure {
       float   someData[1000];
} some_structure;

void demoDataTypeOverflow(int totalItemsCount, some_structure* pItem,
                          int itemPosition) {
 // See http://blogs.msdn.com/oldnewthing/archive/2004/01/29/64389.aspx
 some_structure* data_copy = NULL;
 int bytesToAllocation = totalItemsCount * sizeof(some_structure);
 printf("Bytes to allocation: %d\n", bytesToAllocation);
 data_copy = (some_structure*) malloc(bytesToAllocation);
 if (itemPosition >= 0 && itemPosition < totalItemsCount) {
    memcpy(&(data_copy[itemPosition]), pItem, sizeof(some_structure));
 }
 else {
    printf("Out of bound assignment");
    return;
 }
 free(data_copy);
}
```

# Format string vulnerabilities - motivation

- Quiz – what is insecure in given program?
- Can you come up with attack?

```
int main(int argc, char * argv[]) {
    printf(argv[1]);
    return 0;
}
```

# Format string vulnerabilities

- Wide class of functions accepting format string
  - printf(“%s”, X);
  - resulting string is returned to user (= attacker)
  - formatting string can be under attackers control
  - variables formatted into string can be controlled
- Resulting vulnerability
  - memory content from stack is formatted into string
  - possibly any memory if attacker control buffer pointer

# Information disclosure vulnerabilities

- Exploitable memory vulnerability leading to read access (not write access)
  - attacker learns some information from the memory
- Direct exploitation
  - secret information (cryptographic key, password...)
- Precursor for next step (very important with DEP&ASRL)
  - module version
  - current memory layout after ASRL (stack/heap pointers)
  - stack protection cookies (/GS)

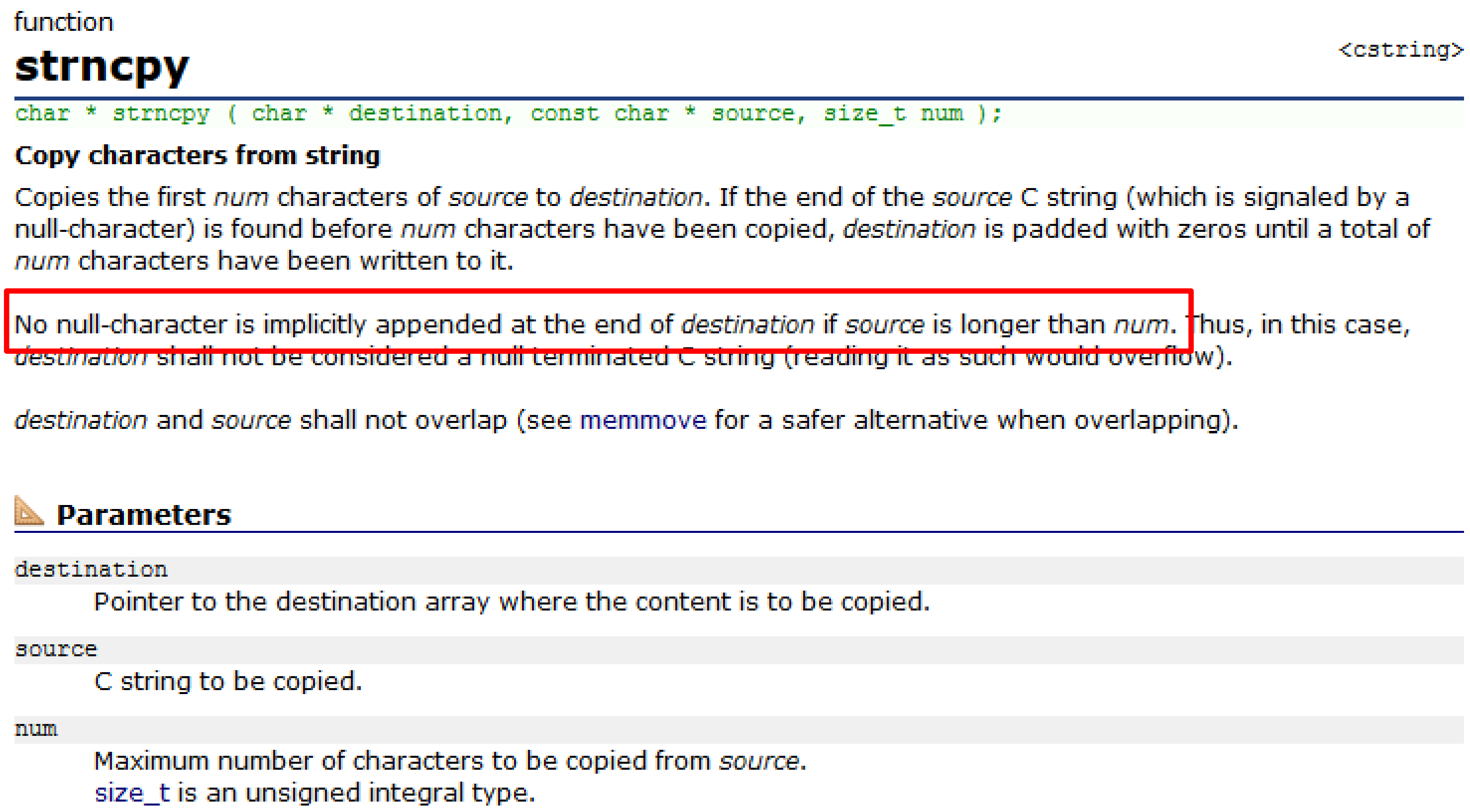
# Format string vulnerability - example

- Example retrieval of security cookie and return address

```
int main(int argc, char* argv[]) {
  char buf[64] = {};
  sprintf(buf, argv[1]);
  return printf("%s\n", buf);
}
```

argv[1] submitted by an attacker
E.g., %x%x%x….%x
Stack content is printed
Including security cookie and RA

# strncpy - manual



function

## strncpy

<cstring>

```
char * strncpy ( char * destination, const char * source, size_t num );
```

**Copy characters from string**

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

*destination* and *source* shall not overlap (see memmove for a safer alternative when overlapping).

### Parameters

`destination`
Pointer to the destination array where the content is to be copied.

`source`
C string to be copied.

`num`
Maximum number of characters to be copied from *source*.
size_t is an unsigned integral type.

*http://www.cplusplus.com/reference/cstring/strncpy/?kw=strncpy*

# Non-terminating functions for strings

- strncpy
- snprintf
- vsnprintf
- mbstowcs
- MultiByteToWideChar

- wcsncpy
- snwprintf
- vsnwprintf
- wcstombs
- WideCharToMultiByte

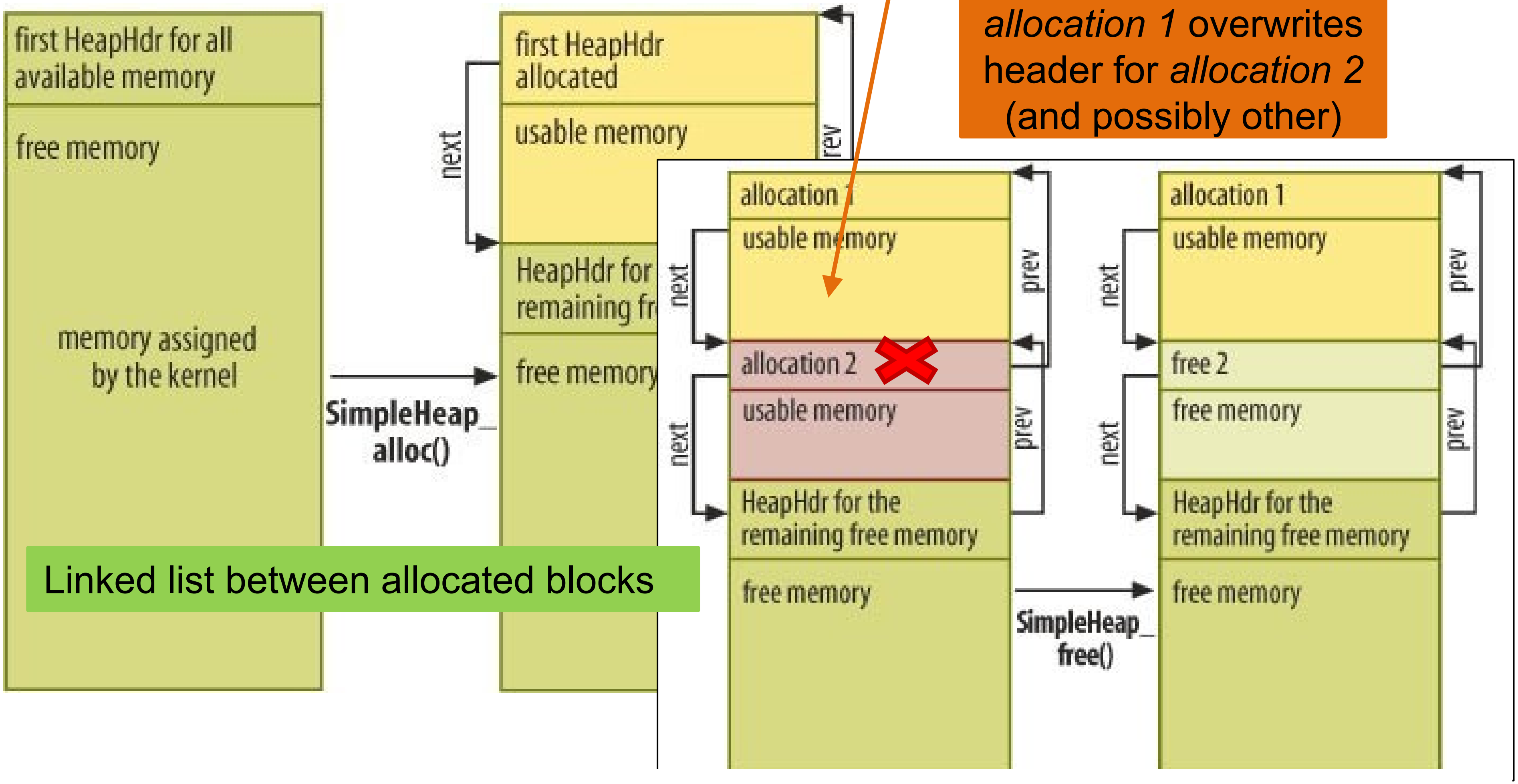
- Non-null terminated Unicode string more dangerous
  - C-string processing stops on first zero
  - any binary zero (ASCII)
  - 16-bit aligned wide zero character (UNICODE)

# Non-terminating functions - example

- What is wrong with following code?

```c
int main(int argc, char* argv[]) {
   char buf[16];
   strncpy(buf, argv[1], sizeof(buf));
   return printf("%s\n",buf);
}
```

# Heap overflow



Buffer overflow in *allocation 1* overwrites header for *allocation 2* (and possibly other)

Linked list between allocated blocks

*Felix "FX" Lindner, http://www.h-online.com/security/features/A-Heap-of-Risk-747220.html*

# Heap overflow – more details

- Assumption: buffer overflow possible for buffer at heap
- Problem:
  - attacker needs to write his pointer to memory later used as jump
  - no return pointer (jump) is stored on heap (as was for stack)
- Different mechanism for misuse
  - overwrite **`malloc`** metadata (few bytes before allocated block)
    - only **`next`**, **`prev`**, **`size`** and **`used`** can be manipulated
    - fake header (**`hdr`**) for fake block is created
  - let **`unlink`** function to be called (merge free blocks)
    - fake block is also merged during merge operation
    - **`hdr->next->next->prev = hdr->next->prev;`**

address in stack that will be interpreted later as jump pointer

address of attacker's code

# SOURCE CODE PREVENTION

# How to detect and prevent problems?

1. Protection on the source code level
   - languages with/without implicit protection
     - containers/languages with array boundary checking
   - usage of safe alternatives to vulnerable function *(this lecture)*
     - vulnerable and safe functions for string manipulations
   - proper input checking
   - automatic detection by static and dynamic checkers
   - security testing, fuzzing
2. Protection by compiler (+ compiler flags) *(this lecture)*
   - runtime checks introduced by compiler (stack protection)
3. Protection by execution environment *(this lecture)*
   - DEP, ASRL...

# How to write code securely (w.r.t. BO) I.

- Be aware of possibilities and principles
- Never trust user's input, always check defensively
- Use safe versions of string/memory functions
- Always provide a format string argument
- Use self-resizing strings (C++ **`std::string`**)
- Use automatic bounds checking if possible
  - C++ **`std::vector.at(i)`** instead of **`vector[i]`**

# How to write code securely (w.r.t. BO) II.

- Run application with lowest possible privileges
- Let your code to be reviewed
- Use compiler-added protection
- Use protection offered by platform (privileges, DEP, ASRL, sandboxing...)

# Secure C library

- Secure versions of commonly misused functions
  - bounds checking for string handling functions
  - better error handling
- Also added to new C standard ISO/IEC 9899:2011
- Microsoft Security-Enhanced Versions of CRT Functions
  - MSVC compiler issue warning C4996, more functions then in C11
- Secure C Library
  - http://docwiki.embarcadero.com/RADStudio/XE3/en/Secure_C_Library
  - http://msdn.microsoft.com/en-us/library/8ef0s5kh%28v=vs.80%29.aspx
  - http://msdn.microsoft.com/en-us/library/wd3wzwts%28v=vs.80%29.aspx
  - http://www.drdobbs.com/cpp/the-new-c-standard-explored/232901670

# Secure C library – selected functions

```
char *gets(
    char *buffer
);

char *gets_s(
    char *buffer,
    size_t sizeInCharacters
);
```

- Formatted input/output functions
  - **gets_s**
  - **scanf_s**, wscanf_s, **fscanf_s**, fwscanf_s, ss vfscanf_s, vfwscanf_s, vscanf_s, vwscanf_s, vswscanf_s
  - **fprintf_s**, fwprintf_s, **printf_s**, printf_s, snprintf_s, snwprintf_s, **sprintf_s**, swprintf_s, vfprintf_s, vfwprintf_s, vprintf_s, vwprintf_s, vsnprintf_s, vsnwprintf_s, vsprintf_s, vswprintf_s
  - functions take additional argument with buffer length
- File-related functions
  - tmpfile_s, tmpnam_s, fopen_s, freopen_s
    - takes pointer to resulting file handle as parameter
    - return error code

# Secure C library – selected functions

- Environment, utilities
  - getenv_s, wgetenv_s
  - bsearch_s, qsort_s
- Memory copy functions
  - memcpy_s, memmove_s, strcpy_s, wcscpy_s, strncpy_s, wcsncpy_s
- Concatenation functions
  - strcat_s, wcscat_s, strncat_s, wcsncat_s
- Search functions
  - strtok_s, wcstok_s
- Time manipulation functions...

# CERT C/C++ Coding Standard

- CERT C Coding Standard
  - https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Coding+Standard
- CERT C++ Coding Standard
  - https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637
- Cern secure coding recommendation for C
  - https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml
- Smashing the stack in 2011
  - https://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/

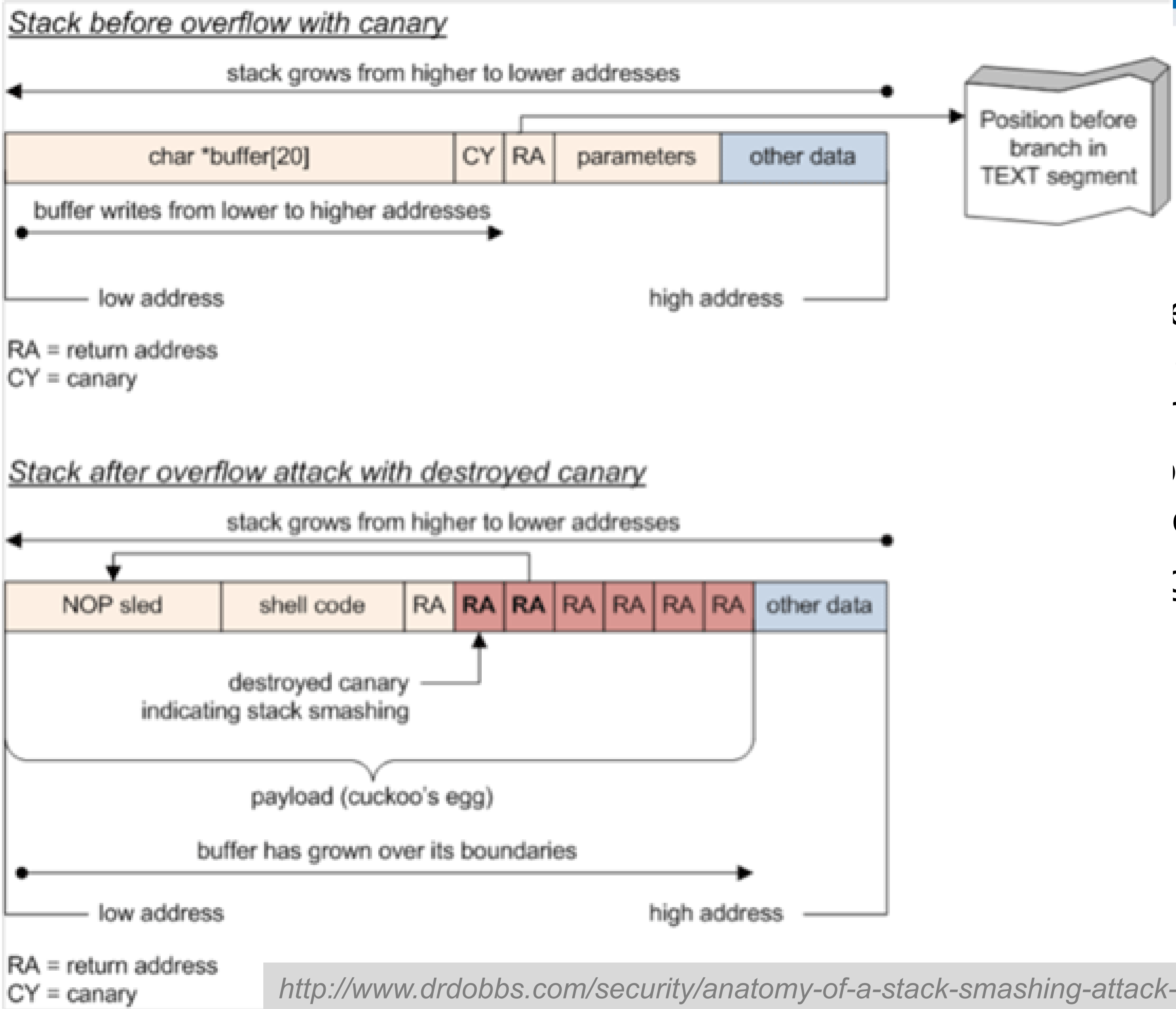
# COMPILER PREVENTIONS

# MSVC Compiler security flags - /RTC

- Microsoft's  MSVC in Visual Studio
  - http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx
- Nice overview of available protections
  - http://msdn.microsoft.com/en-us/library/bb430720.aspx
- Visual Studio → Configuration properties → C/C++ → All options
- Run-time checks
  - /RTCu switch
    - uninitialized variables check
  - /RTCs switch
    - stack protection (stack pointer verification)
    - initialization of local variables to a nonzero value
    - detect overruns and underruns of local variables such as arrays
  - /RTC1 == /RTCsu

# MSVC Compiler security flags - /GS

- /GS switch (added from 2003)
  - http://msdn.microsoft.com/en-us/library/8dbf701c.aspx
  - multiple different protections against buffer overflow
  - mostly focused on stack protection
- /GS protects:
  - return address of function
  - address of exception handler
  - vulnerable function parameters (arguments)
  - some of the local buffers (GS buffers)
- /GS protection is (automatically) added only when needed
  - to limit performance impact, decided by compiler (/GS rules)
  - `#pragma strict_gs_check(on)` - enforce strict rules application


Stack before overflow with canary
stack grows from higher to lower addresses
char *buffer[20]
CY
RA
parameters
other data
Position before branch in TEXT segment
buffer writes from lower to higher addresses
low address
high address
RA = return address
CY = canary
Stack after overflow attack with destroyed canary
stack grows from higher to lower addresses
NOP sled
shell code
RA
RA
RA
RA
RA
RA
RA
other data
destroyed canary indicating stack smashing
payload (cuckoo's egg)
buffer has grown over its boundaries
low address
high address
RA = return address
CY = canary

ed cookie
ocal variables
n address
rolog (add
ookie)
g (check

http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

# /GS Security cookie ('canary') - details

- /GS Security cookie
  - random DWORD number generated at program start
  - master cookie stored in .data section of loaded module
  - xored with function return address (pointer encoding)
  - corruption results in jump to undefined value
- __security_init_cookie
  - http://msdn.microsoft.com/en-us/library/ms235362.aspx

**Stack without /GS**

*Function parameters*
*Function return address*
*Frame pointer*
*Exception Handler frame*
*Locally declared variables and buffers*
*Callee save registers*

**Stack after /GS**

*Function parameters*
*Function return address*
*Frame pointer*
*Cookie*
*Exception Handler frame*
*Locally declared variables and buffers*
*Callee save registers*

# /GS buffers

- Buffers with special protection added
  - http://msdn.microsoft.com/en-us/library/8dbf701c.aspx
  - automatically and heuristically selected by compiler
- Applies to:
  - array larger than 4 bytes, more than two elements, element type is not pointer type
  - data structure with size more than 8 bytes with no pointers
  - buffer allocated by using the _alloca function
    - stack-based dynamic allocation
  - any class or structure with GS buffer

# /GS – vulnerable parameters

- Protection of function’s vulnerable parameters
  - parameters passed into function
  - copy of vulnerable parameters (during fnc’s prolog) placed below the storage area for any other buffers
  - variables prone to buffer overflow are put on higher address so their overflow will not overwrite other local variables
- Applies to:
  - pointer
  - C++ reference
  - C-structure containing pointer
  - GS buffer

# Is /GS protection bulletproof?

**Y**

*Function parameters*
*Function return address (of Y == X)*
*Frame pointer*
*Cookie*
*Exception Handler frame*
*Locally declared variables and buffers*
*Callee save registers*

**X**

*Function parameters*
*Function return address (of X)*
*Frame pointer*
*Cookie*
*Exception Handler frame*
*Locally declared variables and buffers*
*Callee save registers*

- What if pointer to buffer allocated in X is passed into function Y?
  - Return address of X can be overwritten in Y

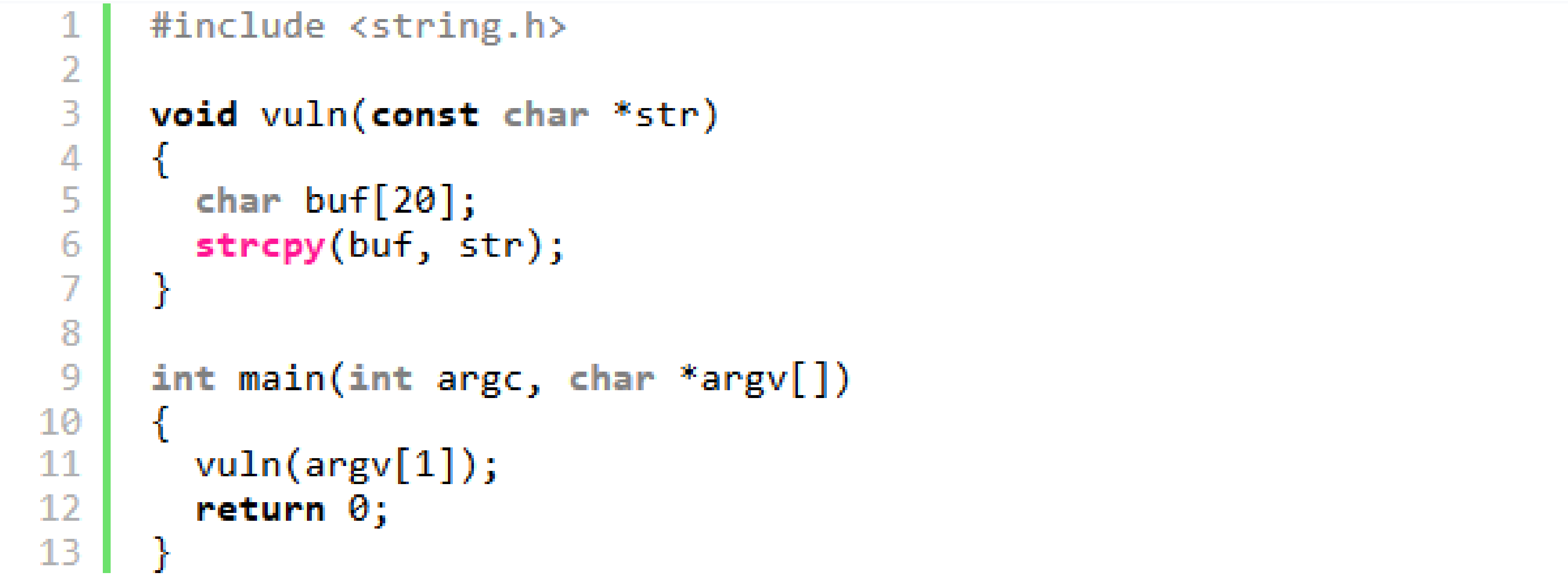
# /GS – what is NOT protected

- /GS compiler option does not protect against all buffer overrun security attacks
- Corruption of address in vtable
  - (table of addresses for virtual methods)
- Example: buffer and a vtable in an object, a buffer overrun could corrupt the vtable
- Functions with variable arguments list (...)

# GCC compiler - StackGuard & ProPolice

- StackGuard released in 1997 as extension to GCC
  - but never included as official buffer overflow protection
- GCC Stack-Smashing Protector (ProPolice)
  - patch to GCC 3.x
  - included in GCC 4.1 release
  - **`-fstack-protector`** (string protection only)
  - **`-fstack-protector-all`** (protection of all types)
  - on some systems enabled by default (OpenBSD)
    - **`-fno-stack-protector`** (disable protection)

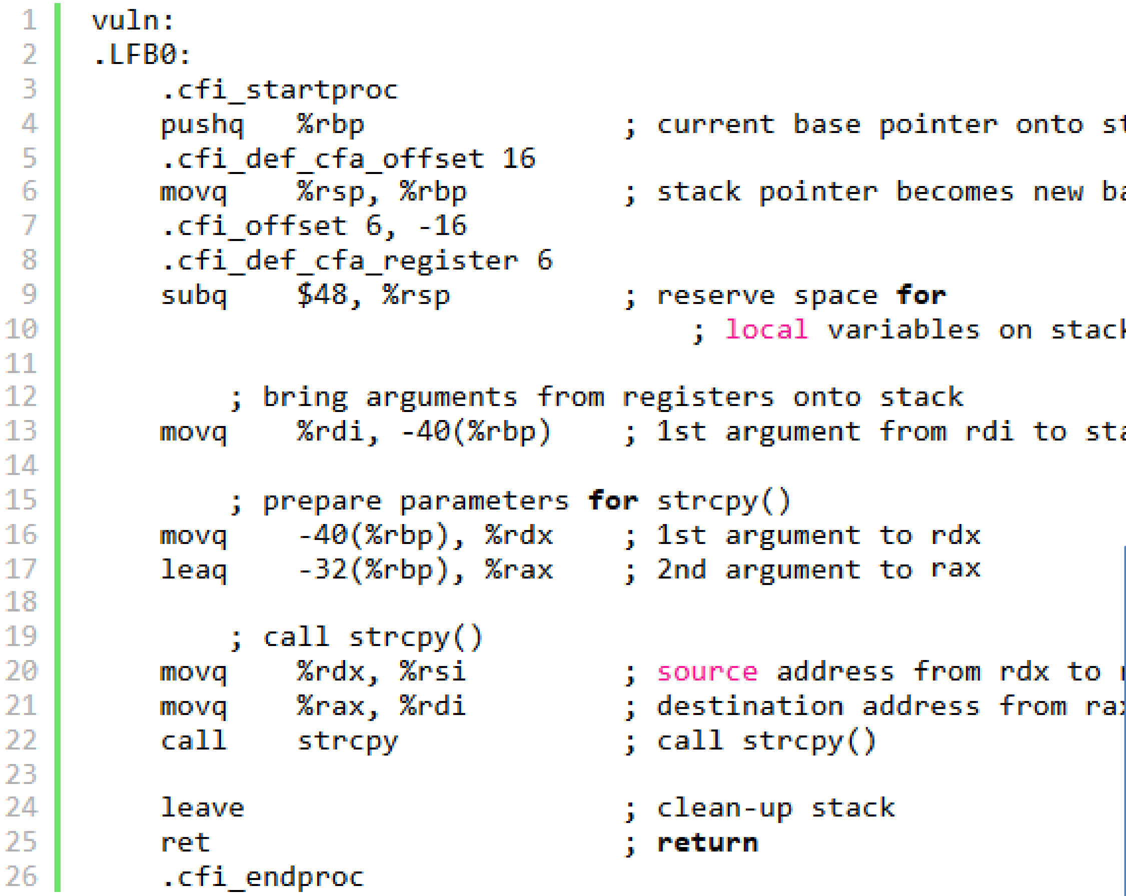
# GCC compiler & ProPolice - example



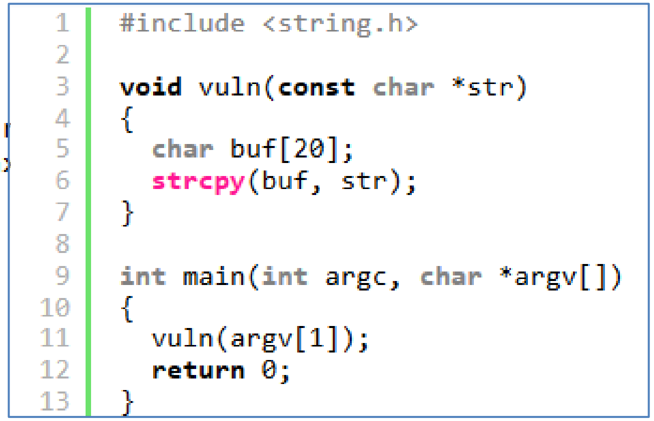
```
#include <string.h>

void vuln(const char *str)
{
  char buf[20];
  strcpy(buf, str);
}

int main(int argc, char *argv[])
{
  vuln(argv[1]);
  return 0;
}
```

*http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#*

# GCC -fno-stack-protector



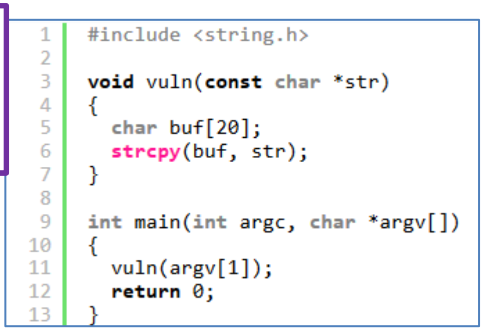
```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                  ; current base pointer onto stack
    .cfi_def_cfa_offset 16
    movq    %rsp, %rbp            ; stack pointer becomes new base pointer
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $48, %rsp             ; reserve space for
                                      ; local variables on stack

        ; bring arguments from registers onto stack
    movq    %rdi, -40(%rbp)       ; 1st argument from rdi to stack

        ; prepare parameters for strcpy()
    movq    -40(%rbp), %rdx       ; 1st argument to rdx
    leaq    -32(%rbp), %rax       ; 2nd argument to rax

        ; call strcpy()
    movq    %rdx, %rsi            ; source address from rdx to r
    movq    %rax, %rdi            ; destination address from ra
    call    strcpy                ; call strcpy()

    leave                         ; clean-up stack
    ret                           ; return
    .cfi_endproc
```



```c
#include <string.h>

void vuln(const char *str)
{
  char buf[20];
  strcpy(buf, str);
}

int main(int argc, char *argv[])
{
  vuln(argv[1]);
  return 0;
}
```

# GCC -fstack-protector-all

```
vuln:
.LFB0:
    .cfi_startproc
    pushq   %rbp                ; current base pointer onto stack
    .cfi_def_cfa_offset 16
    movq    %rsp, %rbp          ; stack pointer becomes new base pointer
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    subq    $48, %rsp           ; reserve space for
                                   ; local variables on stack

        ; bring arguments from registers onto stack
    movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack

        ; SSP's prolog: put canary onto stack
    movq    %fs:40, %rax        ; canary from %fs:40 to rax
    movq    %rax, -8(%rbp)      ; canary from rax onto stack
    xorl    %eax, %eax          ; set rax to zero

        ; prepare parameters for strcpy()
    movq    -40(%rbp), %rdx     ; 1st argument to rdx
    leaq    -32(%rbp), %rax     ; 2nd argument to rax

        ; call strcpy()
    movq    %rdx, %rsi          ; source address from rdx to rsi
    movq    %rax, %rdi          ; destination address from rax to rdi
    call    strcpy              ; call strcpy()

        ; SSP's epilog: check canary
    movq    -8(%rbp), %rax      ; canary from stack to rax
    xorq    %fs:40, %rax        ; original canary XOR rax
    je  .L3                ; if no overflow -> XOR results in zero
                                   ;                => jump to label .L3
                                   ; if overflow    -> XOR results in non-zero
    call    __stack_chk_fail    ;               => call __stack_chk_fail()

.L3:
    leave                       ; clean-up stack
    ret                         ; return
    .cfi_endproc
```



```c
#include <string.h>

void vuln(const char *str)
{
  char buf[20];
  strcpy(buf, str);
}

int main(int argc, char *argv[])
{
  vuln(argv[1]);
  return 0;
}
```

# How to bypass stack protection?

- Scenario:
  - long-term running of daemon on server
  - no exchange of cookie between calls

1. Obtain security cookie by one call
2. Use second call to change only the return address
   - cookie is now known and can be incorporated into stack-smashing data
   - or change cookie so return address will change to attacker target (is xored to return address!)

# Control flow integrity

- Promising technique with low overhead
- Classic CFI (2005), Modular CFI (2014)
  - avg 5% impact, 12% in worst case
  - part of LLVM compiler (but only for C)

1. Analysis of source code to establish control-flow graph (which fnc can call what other fnc)
2. Assign shared labels between valid caller X and callee Y
3. When returning into function X, shared label is checked
4. Return to other function not permitted

https://class.coursera.org/softwaresec-002/lecture/view?lecture_id=49

# PLATFORM PROTECTIONS

# Data Execution Prevention (DEP)

- *Motto: When boundary between code and data blurs (buffer overflow, SQL injection…) then exploitation might be possible.*
- Data Execution Prevention (DEP)
  - – prevents application to execute code from non-executable memory region
  - – available in modern operating systems
    - Linux kernel > 2.6.8, WinXP SP2, Mac OS X, iOS, Android
  - – difference between ‘hardware’ and ‘software’ based DEP

# Hardware DEP

- Supported from AMD64 and Intel Pentium 4
  - OS must add support of this feature (around 2004)
- CPU marks memory page as non-executable
  - most significant bit (63th) in page table entry (NX bit)
  - 0 == execute, 1 == data-only (non-executable)
- Protection typically against buffer overflows
- Cannot protect against all attacks!
  - e.g., code compiled at runtime (produced by JIT compiler) must have both instructions and data in executable page
  - attacker redirect execution to generated code (JIT spray)
  - used to bypass Adobe PDF and Flash security features

# Software DEP

- Unrelated to NX bit (no CPU support required)
- When exception is raised, OS checks if exception handling routine pointer is in executable area
  - Microsoft's Safe Structured Exception Handling
- Software DEP is not preventing general execution in non-executable pages
  - different form of protection than hardware DEP

# Return-oriented programming (ROP) I.

- Return-into-library technique (Solar Designer, 1997)
  - http://seclists.org/bugtraq/1997/Aug/63
  - method for bypassing DEP
  - no write of attacker's code to stack (as is marked by DEP)
  1. function return address is replaced by pointer of selected standard library function instead
  2. library function arguments are also replaced according to attackers needs
  3. function return will result in execution of library function with given arguments
- Example: system call wrappers like `system()`

# Return-oriented programming (ROP) II.

- But 64-bit hardware introduced different calling convention
  - first arguments to function are passed in CPU registers instead of via stack
  - harder to mount return-into-library attack
- Borrowed code chunks
  - attacker tries to find instruction sequences from any function that pop values from the stack into registers
  - necessary arguments are inserted into registers
  - return-into-library attack is then executed as before
- Return-oriented programming extends previous technique
  - multiple borrowed code chunks (gadgets) connected to execute Turing-complete functionality (Shacham, 2007)
  - automated search for gadgets possible by ROPgadget
  - https://www.youtube.com/watch?v=a8_fDdWB2-M
  - partially defended by ASLR (but information leakage)

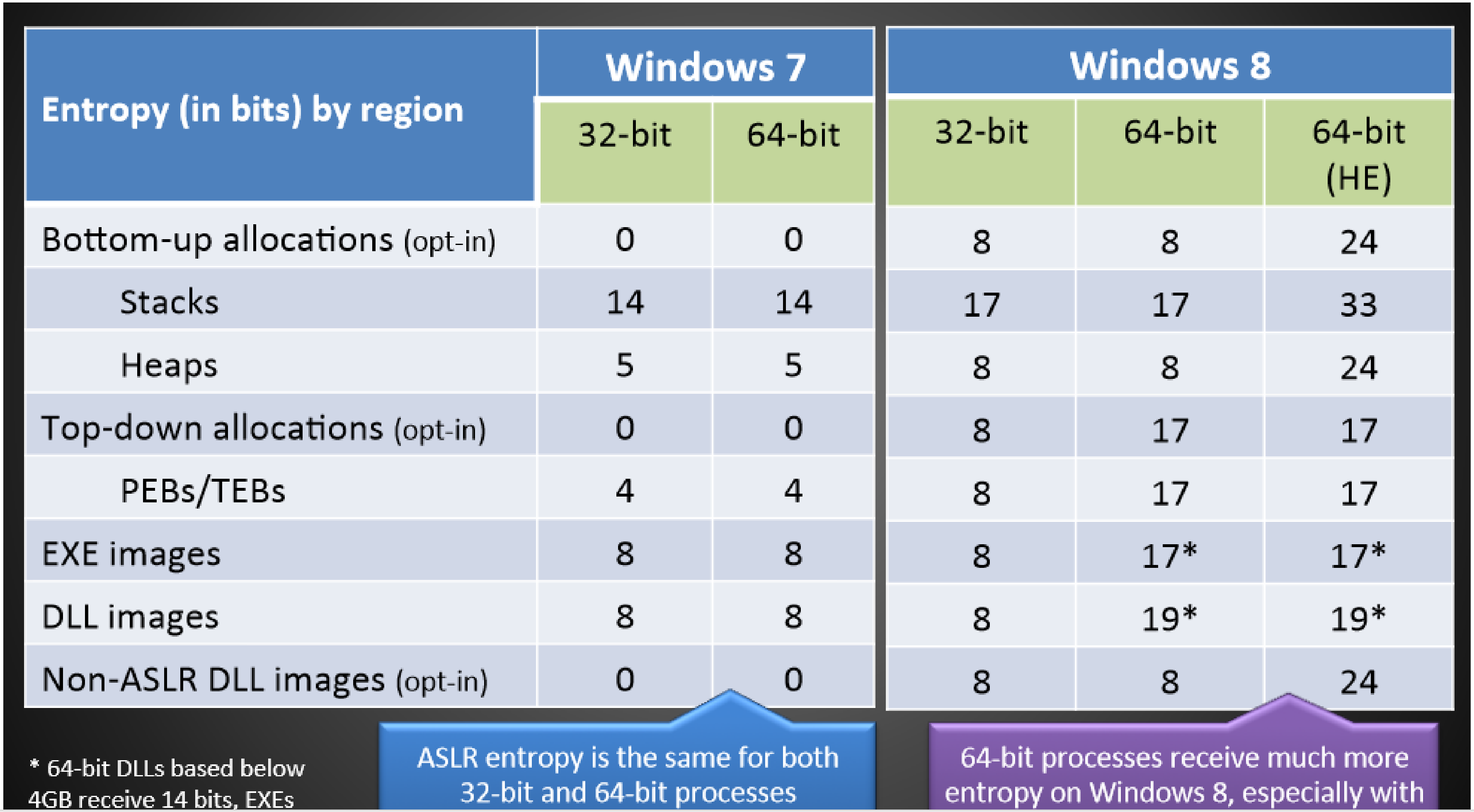
# Address Space Layout Randomization (ASLR)

- Random reposition of executable base, stack, heap and libraries address in process's address space
  - aim is to prevent exploit to reliably jump to required address
  - performed every time a process is executed
  - random offset added to otherwise fixed address
  - entropy of random offset is important (bruteforce)
- Applies to program and also dynamic libraries
- Introduced by Memco software (1997)
  - fully implemented in Linux PaX patch (2001)
  - MS Windows Vista, enabled by default (2007)
  - MS Windows 8, improved entropy (2012)

# ASLR – how much entropy?

- Usually depends on available memory
  - possible attack combination with enforced low-memory situation
- Linux PaX patch (2001)
  - around 24 bits entropy
- MS Windows Vista (2007)
  - heap only around 5-7 bits entropy
  - stack 13-14 bits entropy
  - code 8 bits entropy
  - http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf
- MS Windows 8 (2012)
  - additional entropy, Lagged Fibonacci Generator, registry keys, TPM, Time, ACPI, new rdrand CPU instruction
  - http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

# ASRL entropy in MS Windows  7&8 (2012)



| Entropy (in bits) by region | Windows 7 | | Windows 8 | | |
|---|---|---|---|---|---|
| | 32-bit | 64-bit | 32-bit | 64-bit | 64-bit (HE) |
| Bottom-up allocations (opt-in) | 0 | 0 | 8 | 8 | 24 |
| Stacks | 14 | 14 | 17 | 17 | 33 |
| Heaps | 5 | 5 | 8 | 8 | 24 |
| Top-down allocations (opt-in) | 0 | 0 | 8 | 17 | 17 |
| PEBs/TEBs | 4 | 4 | 8 | 17 | 17 |
| EXE images | 8 | 8 | 8 | 17* | 17* |
| DLL images | 8 | 8 | 8 | 19* | 19* |
| Non-ASLR DLL images (opt-in) | 0 | 0 | 8 | 8 | 24 |

* 64-bit DLLs based below 4GB receive 14 bits, EXEs

ASLR entropy is the same for both 32-bit and 64-bit processes

64-bit processes receive much more entropy on Windows 8, especially with

*Taken from Ken Johnson, Matt Miller (Microsoft Security Engineering Center), BlackHat USA 2012*
*http://media.blackhat.com/bh – us – 12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pd*

# DEP&ASRL – MSVC compilation flags

- /NXCOMPAT (on by default)
  - program is compatible with hardware DEP
- /SAFESEH (on by default, only 32bit programs)
  - software DEP
- /DYNAMICBASE (on by default)
  - basic ASLR
  - Property Pages → Configuration Properties → Linker → Advanced → Randomized Base Address
  - http://msdn.microsoft.com/en-us/library/bb384887.aspx
- /HIGHENTROPYVA (on by default, only 64bit programs)
  - ASRL with higher entropy
  - http://msdn.microsoft.com/en-us/library/dn195771.aspx

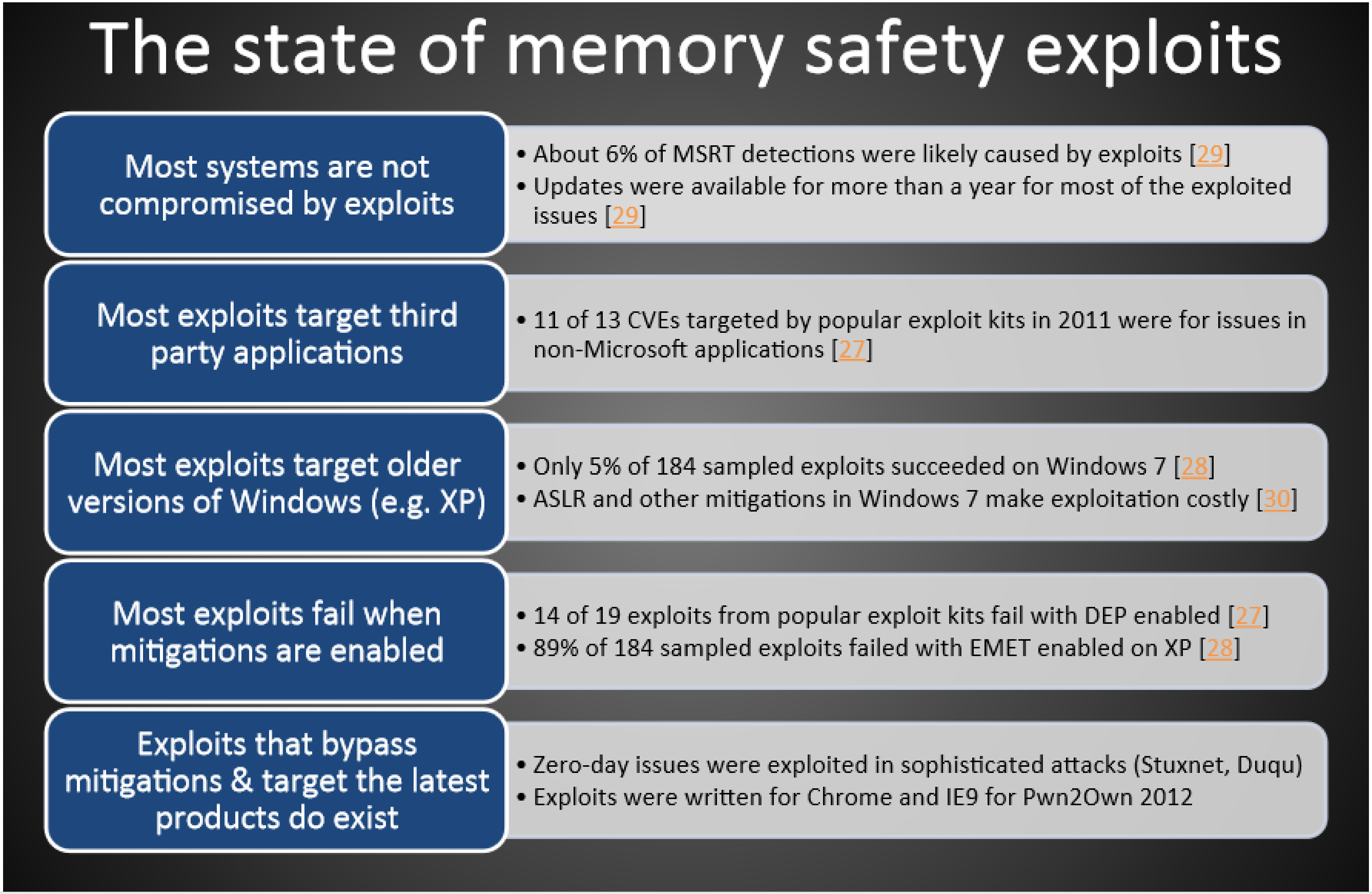
# ASRL – impact on attacks

- ASLR introduced big shift in attacker mentality
- Attacks are now based on gaps in ASRL
  - legacy programs/libraries/functions without ASRL support
    - ! /DYNAMICBASE
  - address space spraying (heap/JIT)
  - predictable memory regions, insufficient entropy

# DEP and ASLR should be combined

- *“For ASLR to be effective, DEP/NX must be enabled by default too.”* M. Howard, Microsoft
- /GS combined with /DYNAMICBASE and /NXCOMPAT
  - /DYNAMICBASE randomizes position of master cookie for /GS
  - /NXCOMPAT prevents insertion of new attackers code and forces ROP
  - /DYNAMICBASE randomizes code chunks used later for ROP
  - /GS prevents modification of return pointer used later for ROP
- Visual Studio → Configuration properties →
  - Linker → All options
  - C/C++ → All options

# SUMMARY



# The state of memory safety exploits

**Most systems are not compromised by exploits**

- About 6% of MSRT detections were likely caused by exploits [29]
- Updates were available for more than a year for most of the exploited issues [29]

**Most exploits target third party applications**

- 11 of 13 CVEs targeted by popular exploit kits in 2011 were for issues in non-Microsoft applications [27]

**Most exploits target older versions of Windows (e.g. XP)**

- Only 5% of 184 sampled exploits succeeded on Windows 7 [28]
- ASLR and other mitigations in Windows 7 make exploitation costly [30]

**Most exploits fail when mitigations are enabled**

- 14 of 19 exploits from popular exploit kits fail with DEP enabled [27]
- 89% of 184 sampled exploits failed with EMET enabled on XP [28]

**Exploits that bypass mitigations & target the latest products do exist**

- Zero-day issues were exploited in sophisticated attacks (Stuxnet, Duqu)
- Exploits were written for Chrome and IE9 for Pwn2Own 2012

*Taken from Ken Johnson, Matt Miller (Microsoft Security Engineering Center), BlackHat USA 2012*
*http://media.blackhat.com/bh – us – 12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pd*

# Final checklist

1. Be aware of possible problems and attacks
   - Don’t make exploitable errors at the first place!
   - Automated protections cannot fully defend everything
2. Use safe versions of vulnerable functions
   - Secure C library (xxx_s functions)
   - Self-resizing strings/containers for C++
3. Compile with all protection flags
   - MSVC: **`/RTC1,/DYNAMICBASE,/GS,/NXCOMPAT`**
   - GCC: **`-fstack-protector-all`**
4. Apply automated tools
   - BinScope Binary Analyzer, static and dynamic analyzers, vulns. scanners
5. Take advantage of protection in the modern OSes
   - and follow news in improvements in DEP, ASRL...

# Mandatory reading

- SANS: 2015 State of Application Security
  - https://www.sans.org/reading-room/whitepapers/analyst/2015-state-application-security-closing-gap-35942
  - What are main differences between builders and defenders?
  - Which applications are of main security concern?
  - Which security standards/methodologies are followed?
- SoK: Eternal War in Memory
  - http://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf
  - http://www.slideshare.net/daniel_bilar/song-2013-so-k-eternal-war-in-memory
  - What are techniques to ensure memory safety?
  - What is performance penalty for memory protection techniques?

Questions

# Additional reading

- Compiler Security Checks In Depth (MS)
  - http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx
- GS cookie effectiveness (MS)
  - http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx
- Design Your Program for Security
  - http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html
- Smashing The Stack For Fun And Profit
  - http://www-inst.cs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- Practical return oriented programming
  - http://365.rsaconference.com/servlet/JiveServlet/previewBody/2573-102-1-3232/RR-304.pdf

# Books - optional

- Writing secure code, chap. 5
- Security Development Lifecycle, chap. 11
- Embedded Systems Security, D., M. Kleidermacher

# Tutorials - optional

- Buffer Overflow Exploitation Megaprimer (Linux)
  - http://www.securitytube.net/groups?operation=view&groupId=4
- Tenouk Buffer Overflow tutorial (Linux)
  - http://www.tenouk.com/Bufferoverflowc/bufferoverflowvulexploitdemo.html
- Format string vulnerabilities primer (Linux)
  - http://www.securitytube.net/groups?operation=view&groupId=3
- Buffer overflow in Easy RM to MP3 utility (Windows)
  - https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

# Heap overflow - references

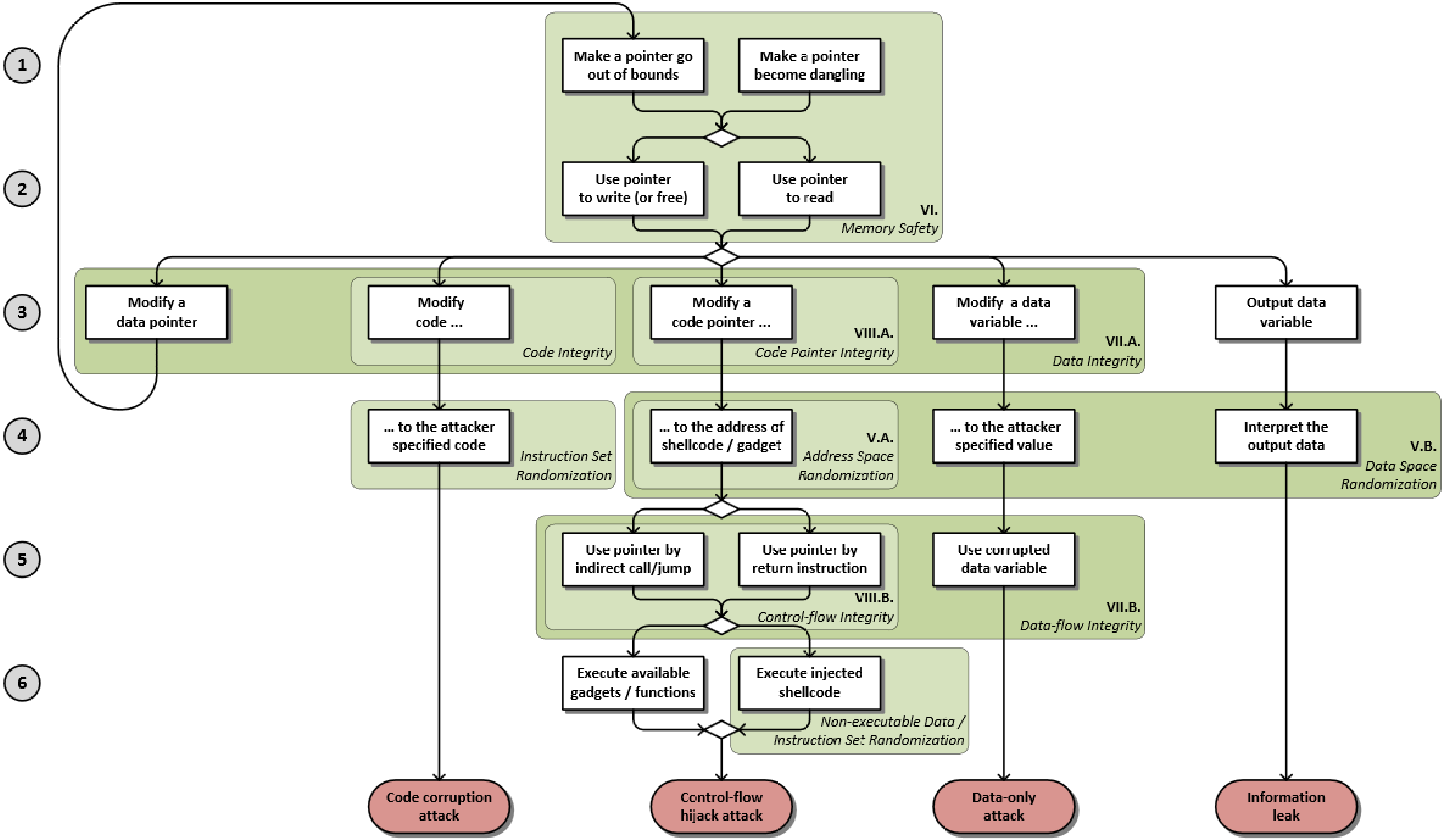
- Detailed explanation (Felix "FX" Lindner, 2006)
  - http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html?view=print
- Explanation in Phrack magazine (blackngel, 2009)
  - http://www.phrack.org/issues.html?issue=66&id=10#article
- Defeating heap protection (Alexander Anisimov)
  - http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf
- Diehard – drop-in replacement for malloc with memory randomization
  - http://plasma.cs.umass.edu/emery/diehard.html
  - https://github.com/emeryberger/DieHard

# ROP - references

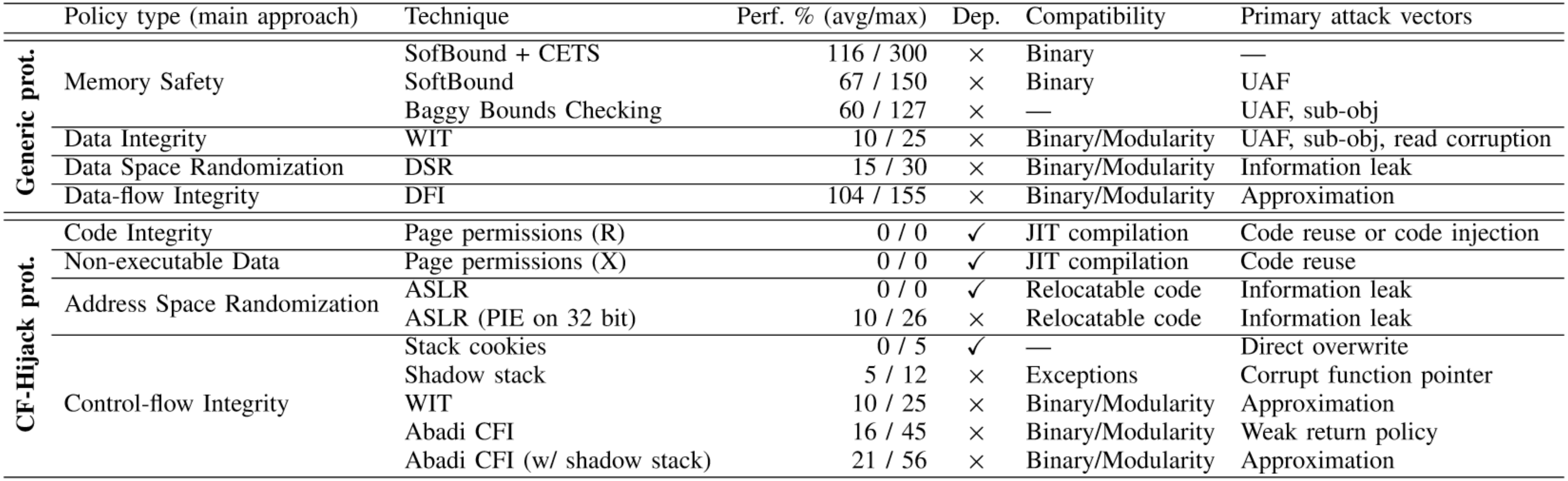
- Explanation of ROP
  - https://www.usenix.org/legacy/event/sec11/tech/full_papers/Schwartz.pdf
- Blind ROP
  - Return-oriented programming without source code
  - http://www.scs.stanford.edu/brop/
- Automatic search for ROP gadgets
  - https://github.com/0vercl0k/rp

http://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

# SoK: Eternal War in Memory


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

# SoK: Eternal War in Memory



| | Policy type (main approach) | Technique | Perf. % (avg/max) | Dep. | Compatibility | Primary attack vectors |
|---|---|---|---|---|---|---|
| Generic prot. | Memory Safety | SofBound + CETS | 116 / 300 | × | Binary | — |
| | | SoftBound | 67 / 150 | × | Binary | UAF |
| | | Baggy Bounds Checking | 60 / 127 | × | — | UAF, sub-obj |
| | Data Integrity | WIT | 10 / 25 | × | Binary/Modularity | UAF, sub-obj, read corruption |
| | Data Space Randomization | DSR | 15 / 30 | × | Binary/Modularity | Information leak |
| | Data-flow Integrity | DFI | 104 / 155 | × | Binary/Modularity | Approximation |
| CF-Hijack prot. | Code Integrity | Page permissions (R) | 0 / 0 | ✓ | JIT compilation | Code reuse or code injection |
| | Non-executable Data | Page permissions (X) | 0 / 0 | ✓ | JIT compilation | Code reuse |
| | Address Space Randomization | ASLR | 0 / 0 | ✓ | Relocatable code | Information leak |
| | | ASLR (PIE on 32 bit) | 10 / 26 | × | Relocatable code | Information leak |
| | Control-flow Integrity | Stack cookies | 0 / 5 | ✓ | — | Direct overwrite |
| | | Shadow stack | 5 / 12 | × | Exceptions | Corrupt function pointer |
| | | WIT | 10 / 25 | × | Binary/Modularity | Approximation |
| | | Abadi CFI | 16 / 45 | × | Binary/Modularity | Weak return policy |
| | | Abadi CFI (w/ shadow stack) | 21 / 56 | × | Binary/Modularity | Approximation |

http://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf