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

What secure programming means?

- Generic good security practices
  - Education, testing, defence in depth, code review…
- Use of secure primitives
  - Random numbers, password handling, secure channel…
- Deployment, maintenance, mitigation
  - Update process, detection of issues in 3rd party libs…
- Usability
  - Hard for users to make a mistake, limit its impact…
- Language-specific issues and procedures
  - Buffer overflow (C/C++), reflection (Java)

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?

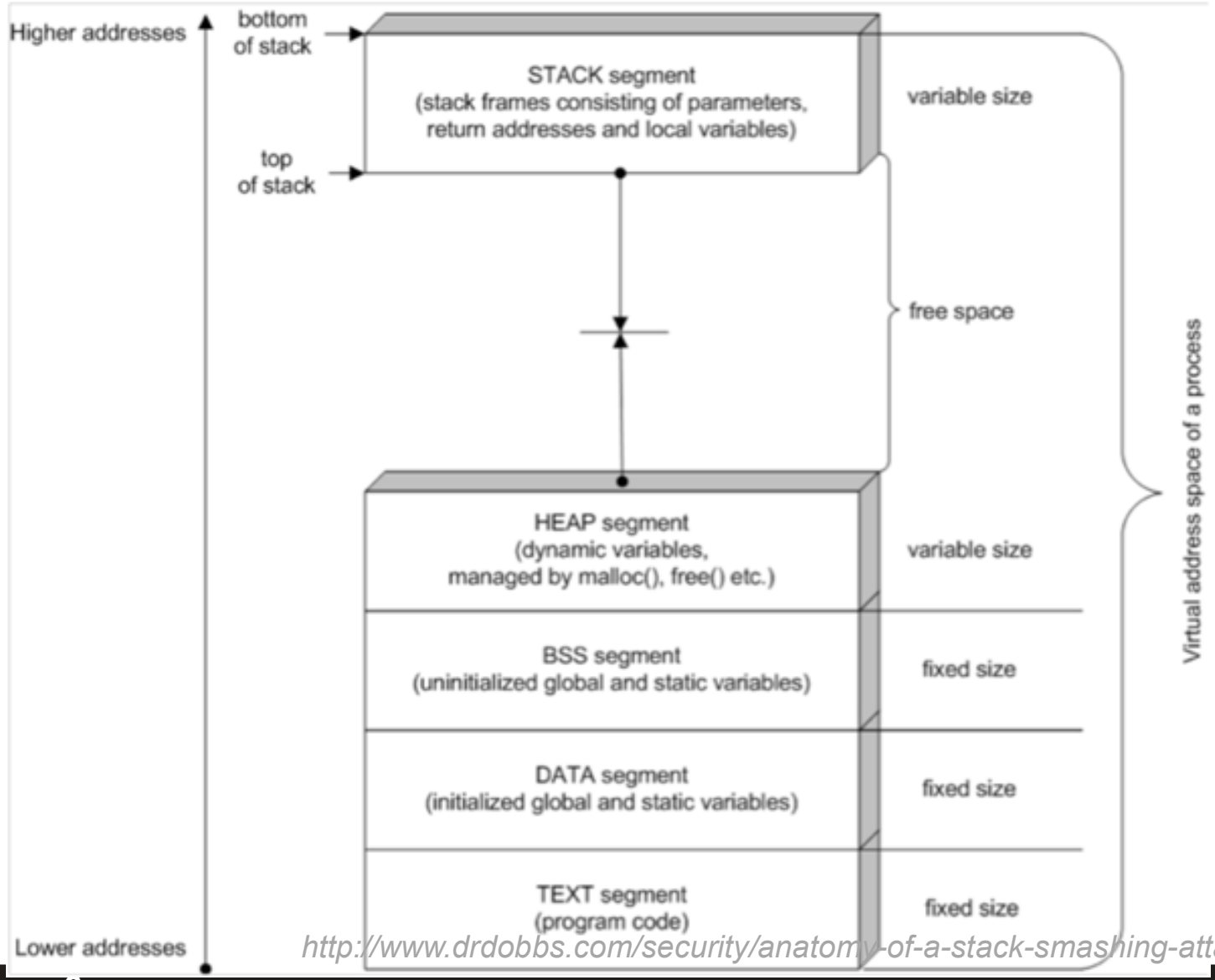
Motivation problem

- 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);
```

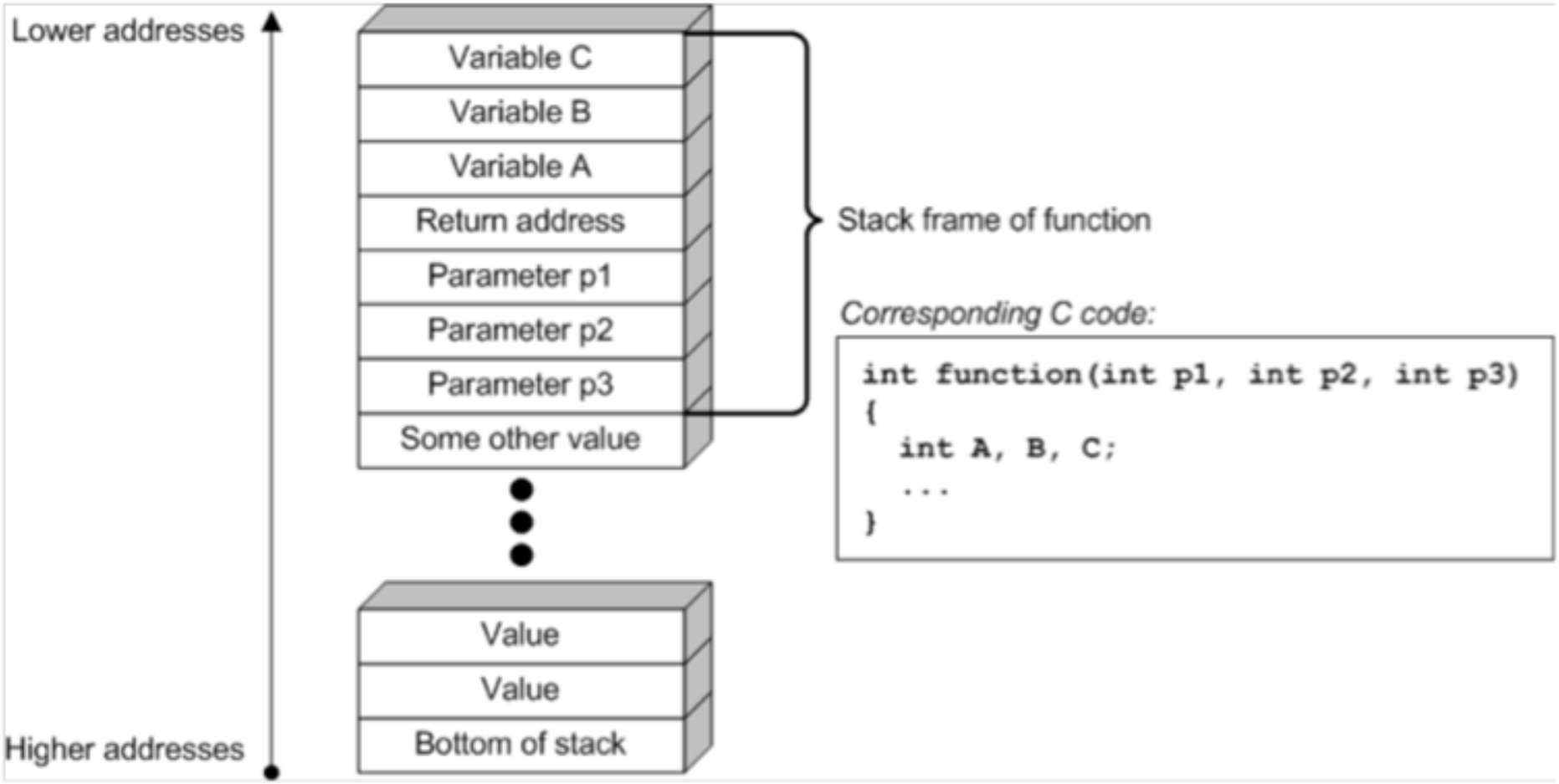
- Classic buffer overflow
- Detailed exploitation demo during labs this week

Process memory layout



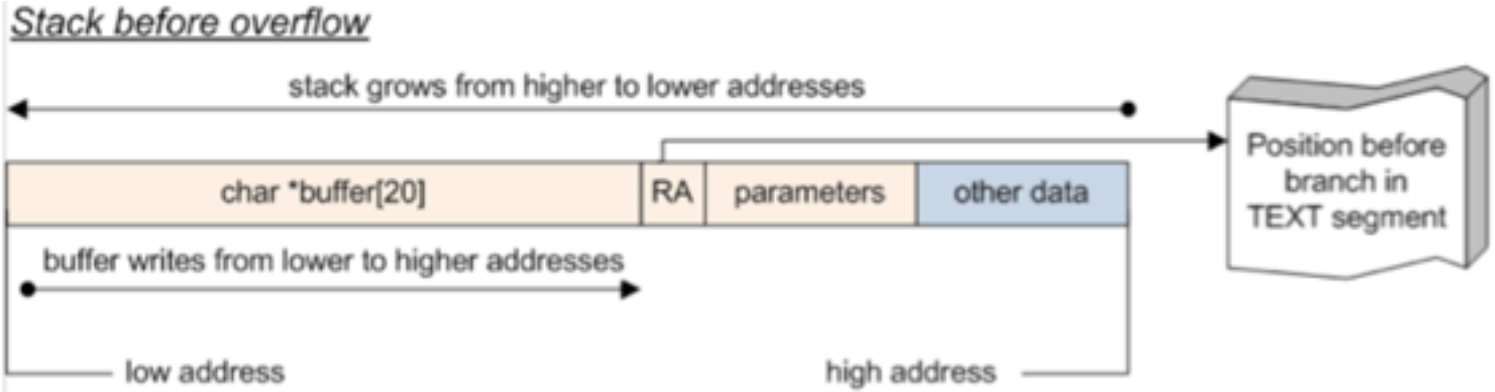
http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#

Stack memory layout



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

Stack overflow



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

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
- Occurs also in higher level languages (Java…)

Make HUGE money with type overflow

- Bitcoin block 74638 (15

Mining block reward
(was 50BTC at 2010, is 12.50BTC now)

Input transaction (with 0.5BTC)
https://blockexplorer.com/tx/237fe8348fc77ace11049931058abb034c99698c7fe99b1cc022b1365a705d39

```
CTransaction(hash=1d5e51, ver=1, vin.size=1, vout.size=2, nLockTime=0
  CTxIn(COutPoint(237fe8, 0), scriptSig=0xA87C02384E1F184B79C6AC)
  CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_H
  CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_H
```

2 output transactions (each with $9*10^{10}$ BTC) !!!

Should have been rejected by miners as value(output) >> value(input), but was not!

Bug dissection

- Bitcoin code uses integer encoding of numbers with fixed position of decimal point (INT64)
  - Smallest fraction of BTC is one Satoshi (sat) = $1/10^8$ BTC
  - 33.54 BTC == 33.54 * 10^8 => 3354000000
- BTW: Why using float numbers is not a good idea
- CTxOut value:92233720368.54275808 BTC
  = 0x7fffffffffff85ee0
- INT64_MAX = 0x7fffffffffffffff
- Sum of 2 CTx = 0xfffffffffffff0bdc0 (overflow)
  = -1000000_{10} = -0.01BTC
  - Difference between input and output is a miner fee

Try this at home!

Type overflow – Bitcoin

```
#include <iostream>
#include <iomanip>
using namespace std;
// Works for Visual Studio compiler, replace __int64 with int64 for other compilers
int main() {
	const __int64 valueMaxInt64 = 0x7fffffffffffffffLL;
	const float COIN = 100000000; // should be __int64 as well, made float for simple printing
	__int64 valueIn = 50000000; // value of input transaction CTxIn
	cout << "CTxIn = " << valueIn / COIN << endl;
	__int64 valueOut1 = 9223372036854275808L; // first out
	cout << "CTxOut1 = " << valueOut1 / COIN << endl;
	__int64 valueOut2 = 9223372036854275808L; // second out
	cout << "CTxOut2 = " << valueOut2 / COIN << endl;

	__int64 valueOutSum = valueOut1 + valueOut2;   // sum which overflow
	cout << "CTxOut sum = " << valueOutSum / COIN << endl;
	// Difference between input and output is interpreted as fee for a miner (0.01 BTC)
	__int64 fee = valueIn - valueOutSum;
	cout << "Miner fee = " << fee / COIN << endl;
	return 0;
}
```

Bug impact (CVE-2010-5139)

- 2 * 92233720368.54275808 + 0.01 BTC artificially created in single transaction
- Detected 1.5 hours after the transaction occurred
- Code patched and blockchain hard forked to abandon branch with malicious transaction
  - Hard fork was possible in early days of Bitcoin, would be more difficult now
  - BTW: Ethereum had hard fork after $60M DAO hack
- https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5139
- https://bitcointalk.org/index.php?topic=822.0

BugFix – proper checking for overflow

https://github.com/bitcoin/bitcoin/commit/d4c6b90ca3f9b47adb1b2724a0c3514f80635c84#diff-118fcbaaba162ba17933c7893247df3aR1013

11 main.h — View

```
@@ -18,6 +18,7 @@ static const unsigned int MAX_SIZE = 0x02000000;
 static const unsigned int MAX_BLOCK_SIZE = 1000000;
 static const int64 COIN = 100000000;
 static const int64 CENT = 1000000;
+static const int64 MAX_MONEY = 21000000 * COIN;
 static const int COINBASE_MATURITY = 100;

 static const CBigNum bnProofOfWorkLimit(~uint256(0) >> 32);
@@ -471,10 +472,18 @@ class CTransaction
         if (vin.empty() || vout.empty())
             return error("CTransaction::CheckTransaction() : vin or vout empty");

-        // Check for negative values
+        // Check for negative or overflow output values
+        int64 nValueOut = 0;
         foreach(const CTxOut& txout, vout)
+        {
             if (txout.nValue < 0)
                 return error("CTransaction::CheckTransaction() : txout.nValue negative");
+            if (txout.nValue > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout.nValue too high");
+            nValueOut += txout.nValue;
+            if (nValueOut > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout total too high");
+        }

         if (IsCoinBase())
         {
```

Try this at home!

Questions

- When exactly overflow happens?
- Why mining reward was 50.51 and not exactly 50?
  - CTxOut(nValue= 50.51000000
- How to check for type overflow?

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 assi
    return;
 }
 free(data_copy);
}
```

Basic idea:

- Data to be copied into newly allocated mem.
- Computation of required size type-overflow
- Too small memory chunk is allocated
- Copy will write behind allocated memory

Safe add and mult operations in C/C++

- Compiler-specific non-standard extensions of C/C++
- GCC: __builtin_add_overflow, __builtin_mul_overflow …

  bool __builtin_add_overflow **(**type1 a**,** type2 b**,** type3 ***res)**
  - Result returned as third (pointer passed) argument
  - Returns true if overflow occurs
  - https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html
- MSVC: SafeInt wrapper template (for int, char…)
  - Overloaded all common operations (drop in replacement)
  - Returns SafeIntException if overflow/underflow
  - https://msdn.microsoft.com/en-us/library/dd570023.aspx

```
#include <safeint.h>
using namespace msl::utilities;                    // Normal use
SafeInt<int> c1 = 1; SafeInt<int> c2 = 2;           c1 = c1 + c2;
```

Safe add and mult operations in Java

- Java SE 8 introduces extensions to java.lang.Math
- ArithmeticException thrown if overflow/underflow

```
public static int addExact(int x, int y)
public static long addExact(long x, long y)
public static int decrementExact(int a)
public static long decrementExact(long a)
public static int incrementExact(int a)
public static long incrementExact(long a)
public static int multiplyExact(int x, int y)
public static long multiplyExact(long x, long y)
public static int negateExact(int a)
public static long negateExact(long a)
public static int subtractExact(int x, int y)
public static long subtractExact(long x, long y)
public static int toIntExact(long value)
```

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 (= potential 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&ASLR)
  - module version
  - current memory layout after ASLR (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]);
  printf("%s\n", buf);
  return 0;
}
```

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

Don’t let user/attacker to provide own formatting strings

Non-terminating functions - example

- What is wrong with following code?

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

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.

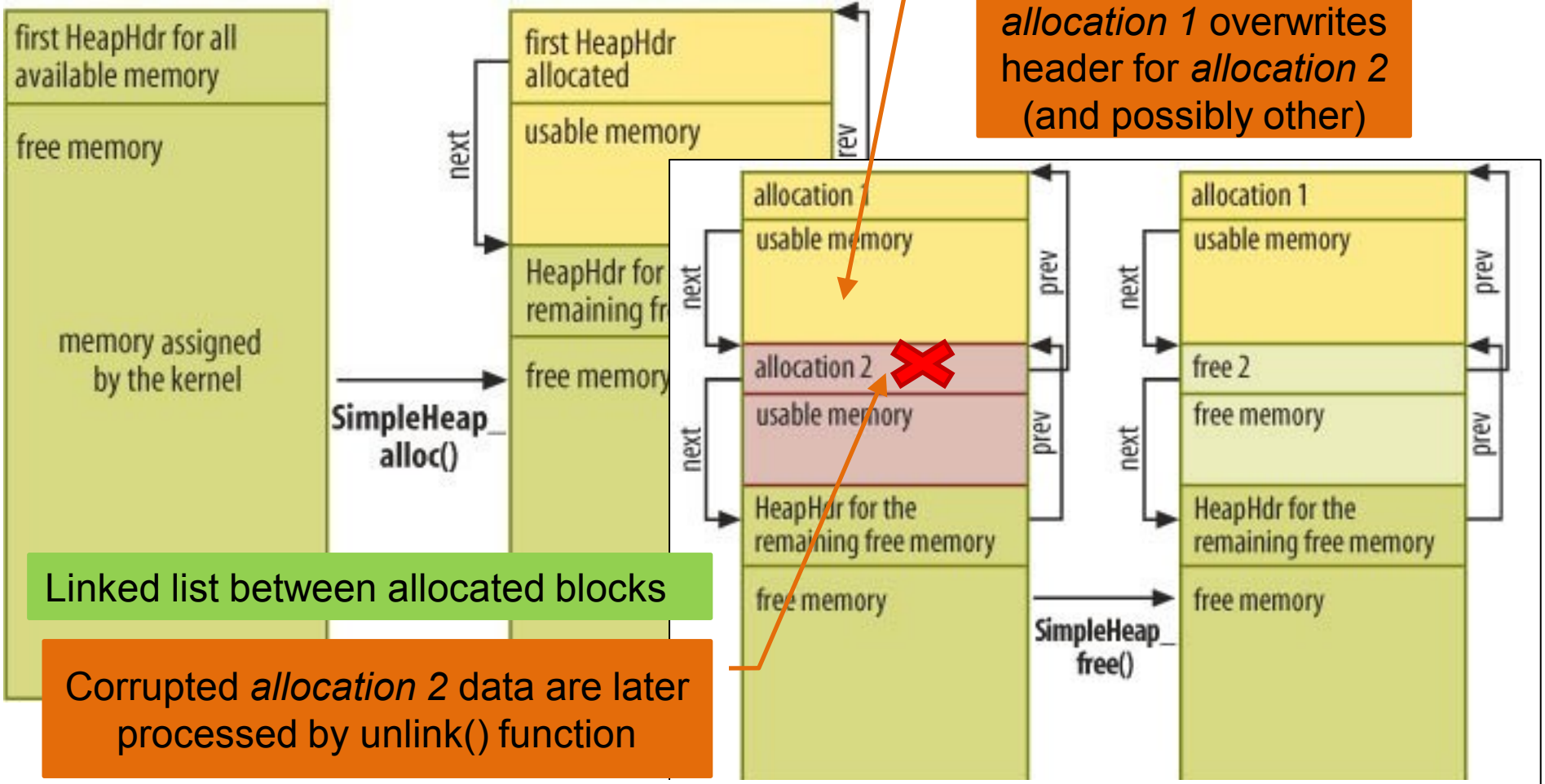
http://www.cplusplus.com/reference/cstring/strncpy/?kw=strncpy

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)

Heap overflow



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 PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS

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 (*next lectures*)
   - automatic detection by static and dynamic checkers (*next lectures*)
   - Code review, security testing (*next lectures*)
2. Protection by compiler (+ compiler flags) *(this lecture)*
   - runtime checks introduced by compiler (stack protection)
3. Protection by execution environment *(this lecture)*
   - DEP, ASLR...

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

- Be aware of possibilities and principles
- Use language with array boundary checks
- 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, ASLR, 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...

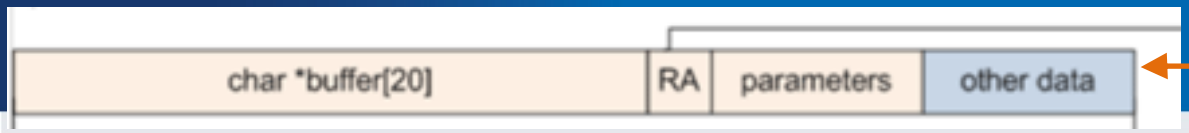
CERT C/C++ Coding Standard

- 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/

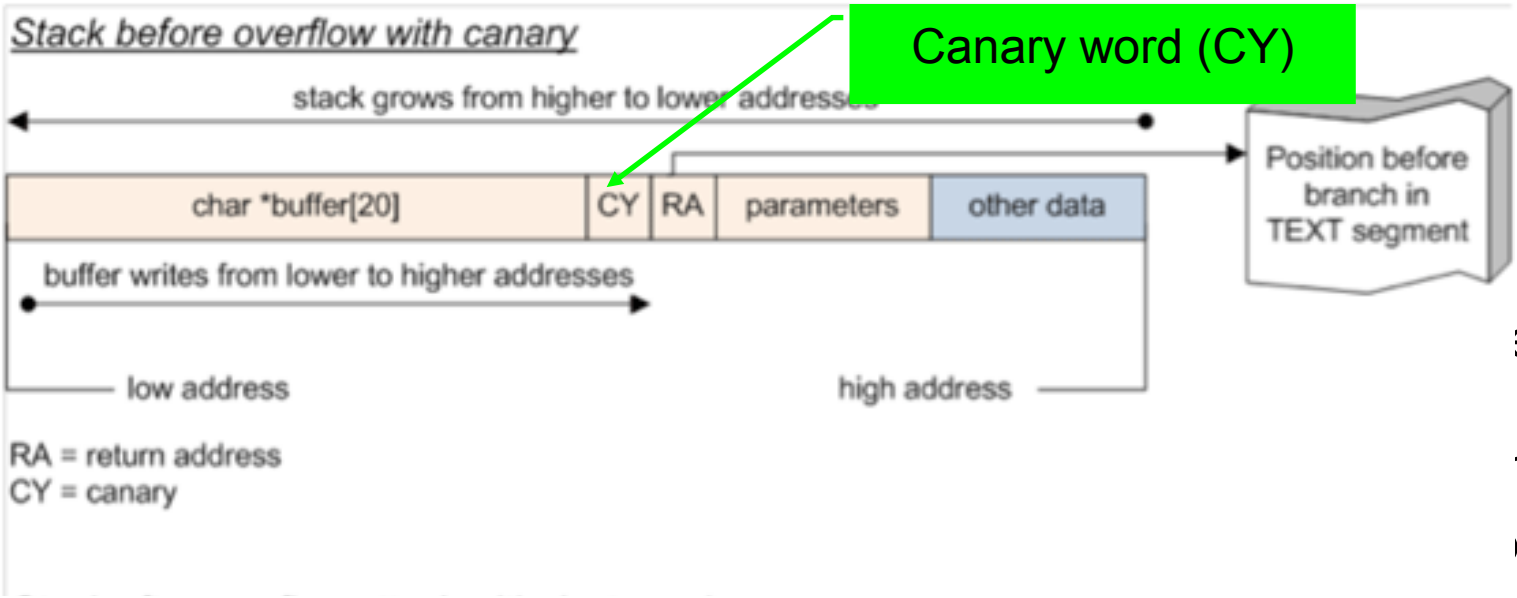
SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS



Stack without canary word



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#

RA = return address

MSVC Compiler security flags - /GS

- /GS switch (added from 2003, evolves in time)
  - 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

/GS is applied in both DEBUG and RELEASE modes

/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

```
1  #include <string.h>
2
3  void vuln(const char *str)
4  {
5    char buf[20];
6    strcpy(buf, str);
7  }
8
9  int main(int argc, char *argv[])
10 {
11   vuln(argv[1]);
12   return 0;
13 }
```

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

GCC -fno-stack-protector

```
1  vuln:
2  .LFB0:
3      .cfi_startproc
4      pushq   %rbp                ; current base pointer onto stack
5      .cfi_def_cfa_offset 16
6      movq    %rsp, %rbp          ; stack pointer becomes new base pointer
7      .cfi_offset 6, -16
8      .cfi_def_cfa_register 6
9      subq    $48, %rsp           ; reserve space for
10                                     ; local variables on stack
11
12         ; bring arguments from registers onto stack
13     movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack
14
15         ; prepare parameters for strcpy()
16     movq    -40(%rbp), %rdx     ; 1st argument to rdx
17     leaq    -32(%rbp), %rax     ; 2nd argument to rax
18
19         ; call strcpy()
20     movq    %rdx, %rsi          ; source address from rdx to r
21     movq    %rax, %rdi          ; destination address from rax
22     call    strcpy              ; call strcpy()
23
24     leave                       ; clean-up stack
25     ret                         ; return
26     .cfi_endproc
```

```
1  #include <string.h>
2
3  void vuln(const char *str)
4  {
5    char buf[20];
6    strcpy(buf, str);
7  }
8
9  int main(int argc, char *argv[])
10 {
11   vuln(argv[1]);
12   return 0;
13 }
```

GCC -fstack-protector-all

```
1  vuln:
2  .LFB0:
3      .cfi_startproc
4      pushq   %rbp                ; current base pointer onto stack
5      .cfi_def_cfa_offset 16
6      movq    %rsp, %rbp          ; stack pointer becomes new base pointer
7      .cfi_offset 6, -16
8      .cfi_def_cfa_register 6
9      subq    $48, %rsp           ; reserve space for
10                                     ; local variables on stack
11
12         ; bring arguments from registers onto stack
13     movq    %rdi, -40(%rbp)     ; 1st argument from rdi to stack
14
15         ; SSP's prolog: put canary onto stack
16     movq    %fs:40, %rax        ; canary from %fs:40 to rax
17     movq    %rax, -8(%rbp)      ; canary from rax onto stack
18     xorl    %eax, %eax          ; set rax to zero
19
20         ; prepare parameters for strcpy()
21     movq    -40(%rbp), %rdx     ; 1st argument to rdx
22     leaq    -32(%rbp), %rax     ; 2nd argument to rax
23
24         ; call strcpy()
25     movq    %rdx, %rsi          ; source address from rdx to rsi
26     movq    %rax, %rdi          ; destination address from rax to rdi
27     call    strcpy              ; call strcpy()
28
29         ; SSP's epilog: check canary
30     movq    -8(%rbp), %rax      ; canary from stack to rax
31     xorq    %fs:40, %rax        ; original canary XOR rax
32     je  .L3                 ; if no overflow -> XOR results in zero
33                                ;                   => jump to label .L3
34                                ; if overflow    -> XOR results in non-zero
35     call    __stack_chk_fail    ;                => call __stack_chk_fail()
36
37 .L3:
38     leave                       ; clean-up stack
39     ret                         ; return
40     .cfi_endproc
```

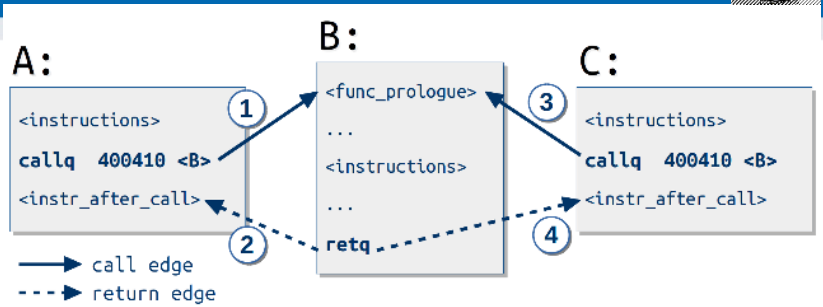
```
1  #include <string.h>
2
3  void vuln(const char *str)
4  {
5    char buf[20];
6    strcpy(buf, str);
7  }
8
9  int main(int argc, char *argv[])
10 {
11   vuln(argv[1]);
12   return 0;
13 }
```

How to bypass stack protection cookie?

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

1. Obtain security cookie by one call
   - cookie is now known and can be incorporated into stack-smashing data
2. Use second call to change only the 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 C compiler (CFI usable for other languages as well)

1. Analysis of source code to establish control-flow graph (which function can call what other functions)
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 is not permitted

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

https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-carlini.pdf

SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

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 > 2.6.8, WinXPSP2, Mac OSX, 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 prevented 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 loaded into memory
  - random offset added to otherwise fixed address
  - applies to program and also dynamic libraries
  - entropy of random offset is important (bruteforce)
- Operating System kernel ASLR (kASLR)
  - more problematic as long-running (random, but fixed until reboot)
- Introduced by Memco software (1997)
  - fully implemented in Linux PaX patch (2001)
  - MS Vista, enabled by default (2007), MS Win 8 more 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

ASLR 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

(Microsoft Security Engineering Center), BlackHat USA 2012

ASLR – impact on attacks

- ASLR introduced big shift in attacker mentality
- Attacks are now based on gaps in ASLR
  - legacy programs/libraries/functions without ASLR 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
  - /NXCOMPAT (==DEP)
  - prevents insertion of new attackers code and forces ROP
  - /DYNAMICBASE (==ASLR) randomizes code chunks utilized by ROP
  - /GS prevents modification of return pointer used later for ROP
  - /DYNAMICBASE randomizes position of master cookie for /GS
- 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

(Microsoft Security Engineering Center), BlackHat USA 2012

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, ASLR...

Mandatory reading

- SANS: 2016 State of Application Security
  - https://www.sans.org/reading-room/whitepapers/analyst/2016-state-application-security-skills-configurations-components-36917
  - How mature is AppSec in companies? Which industry leads?
  - Which applications are of main security concern?
  - What is expected time to deploy patch for security vulnerability?
  - What about third-party components?
- 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

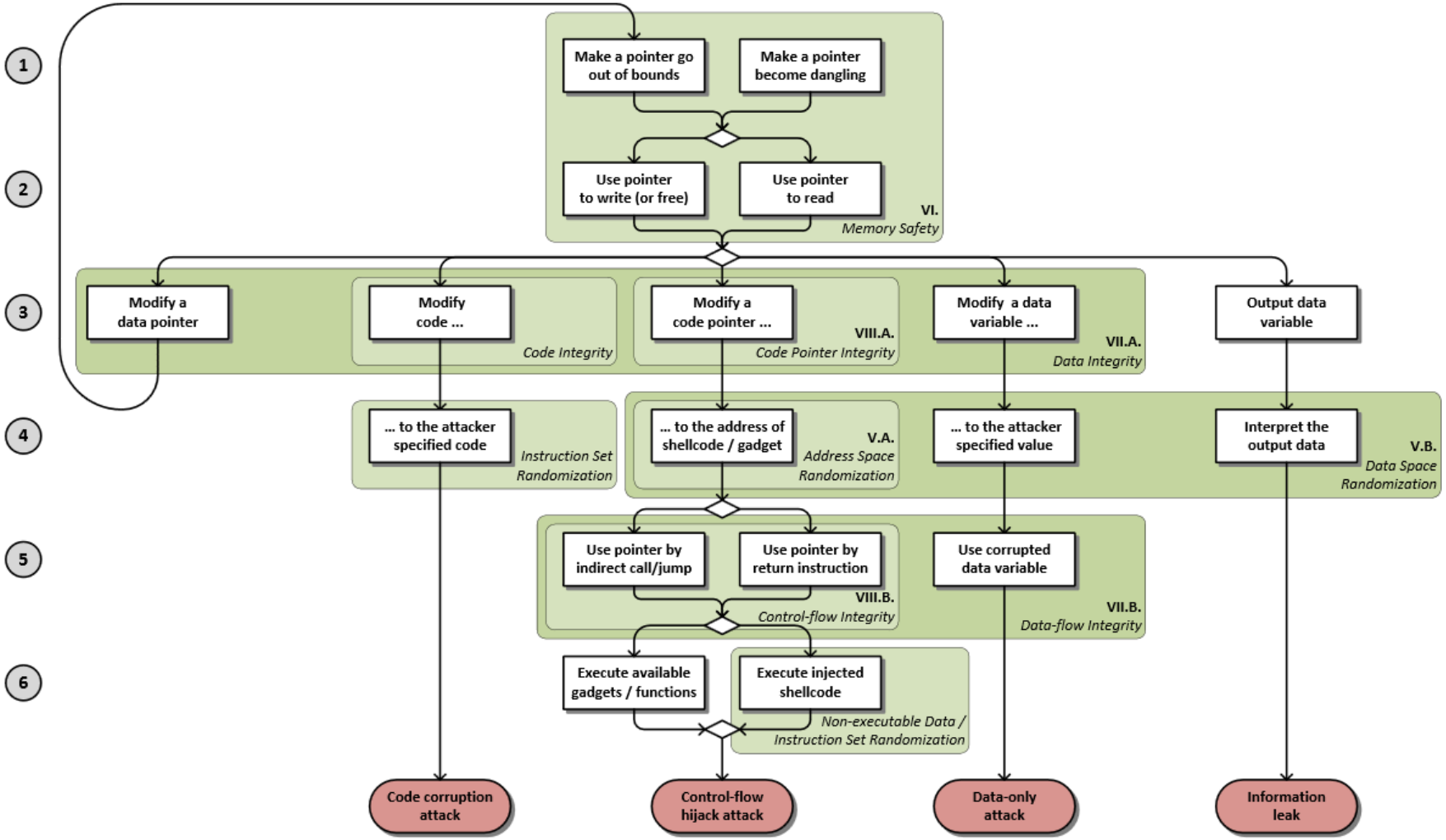
- Detailed explanation (Felix "FX" Lindner, 2006)
  - http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html?view=print
- Explanation in Phrack magazine (blackngel, 2009)
  - http://www.phrack.org/issues.html?issue=66&id=10#article
- Defeating heap protection (Alexander Anisimov)
  - http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.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



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