PA193 - Secure coding principles and practices

Language level vulnerabilities:
Buffer overflow, type overflow, strings

Petr Švenda *svenda@fi.muni.cz* *@rngsec*
Centre for Research on Cryptography and Security, Masaryk University

COURSE TRIVIA:
PA193_00_COURSE_ORGANISATION_2019.PPTX

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)

PROBLEM?

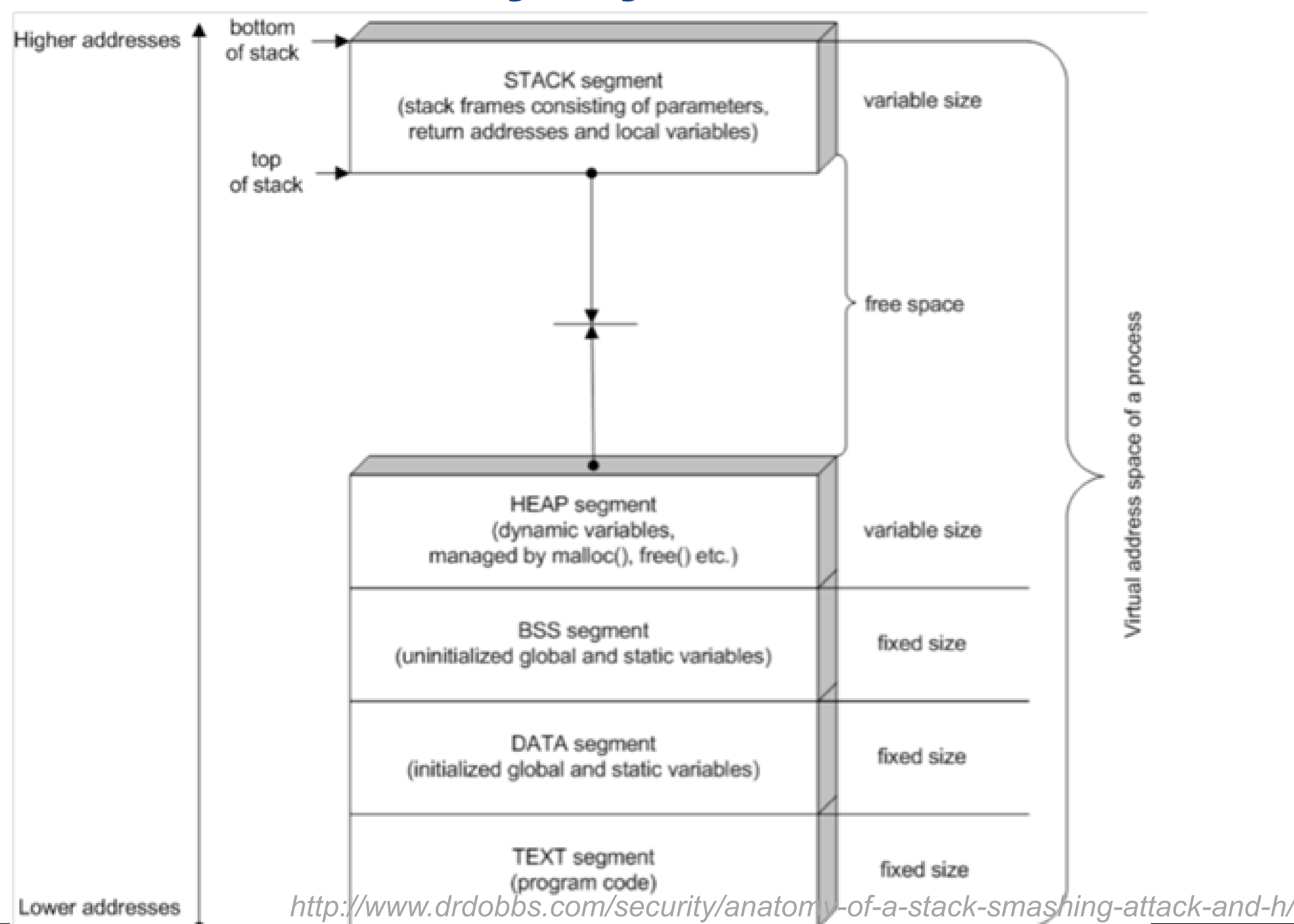
Motivation problem

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

```
#define USER_INPUT_MAX_LENGTH  20
char buffer[USER_INPUT_MAX_LENGTH];
bool  isAdmin = false;
gets(buffer);
```

- 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

Lower addresses

Variable C
Variable B
Variable A
Return address
Parameter p1
Parameter p2
Parameter p3
Some other value
⋮
Value
Value
Bottom of stack

Higher addresses

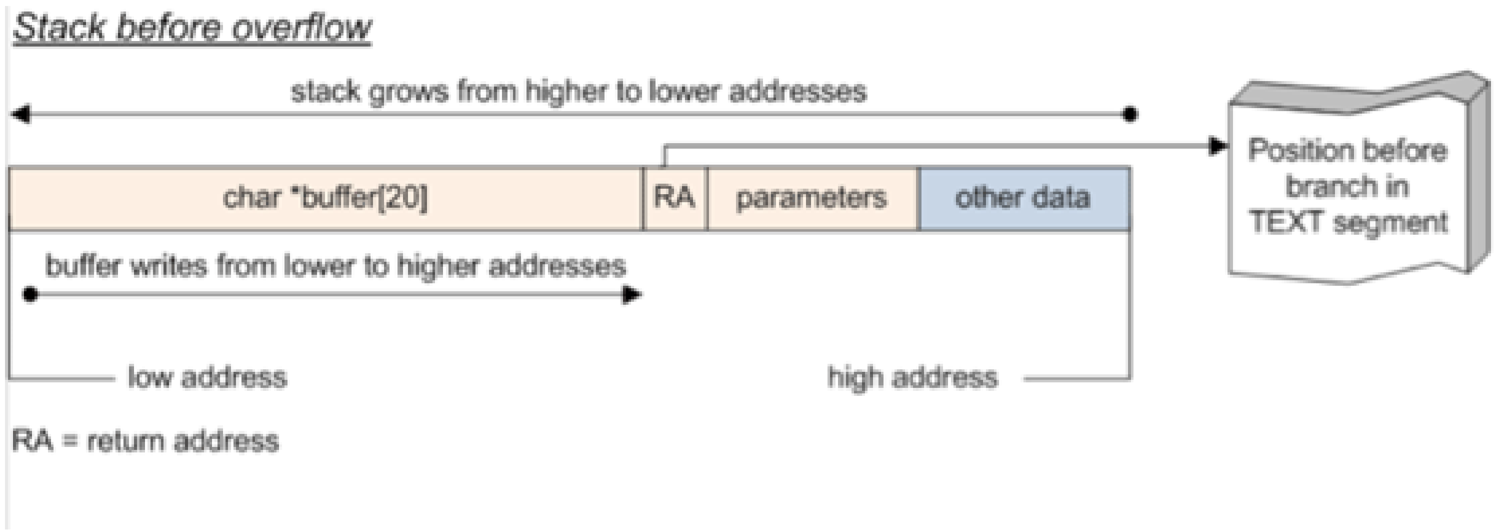
Stack frame of function

Corresponding C code:

```
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



RA = return address

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

https://en.wikipedia.org/wiki/Memory_safety

Types of memory errors [edit]

Many different types of memory errors can occur:[18][19]

- **Access errors**: invalid read/write of a pointer
  - **Buffer overflow** - out-of-bound writes can corrupt the content of adjacent objects, or internal data (like bookkeeping information for the heap) or return addresses.
  - **Buffer over-read** - out-of-bound reads can reveal sensitive data or help attackers bypass address space layout randomization.
  - **Race condition** - concurrent reads/writes to shared memory
  - **Invalid page fault** - accessing a pointer outside the virtual memory space. A null pointer dereference will often cause an exception or program termination in most environments, but can cause corruption in operating system kernels or systems without memory protection, or when use of the null pointer involves a large or negative offset.
  - **Use after free** - dereferencing a dangling pointer storing the address of an object that has been deleted.
- **Uninitialized variables** - a variable that has not been assigned a value is used. It may contain an undesired or, in some languages, a corrupt value.
  - **Null pointer dereference** - dereferencing an invalid pointer or a pointer to memory that has not been allocated
  - **Wild pointers** arise when a pointer is used prior to initialization to some known state. They show the same erratic behaviour as dangling pointers, though they are less likely to stay undetected.
- **Memory leak** - when memory usage is not tracked or tracked incorrectly
  - **Stack exhaustion** - occurs when a program runs out of stack space, typically because of too deep recursion. A guard page typically halts the program, preventing memory corruption, but functions with large stack frames may bypass the page.
  - **Heap exhaustion** - the program tries to allocate more memory than the amc allocation.
  - **Double free** - repeated calls to free may prematurely free a new object at th especially in allocators that use free lists.
  - **Invalid free** - passing an invalid address to free can corrupt the heap.
  - **Mismatched free** - when multiple allocators are in use, attempting to free memory with a dealloc
  - **Unwanted aliasing** - when the same memory location is allocated and modified twice for unrelat

Are other languages also affected by memory overflow vulnerabilities?
Is Java, Python… affected?

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 (15th August

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

ashMerk

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_HASH160 0xB
  CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0x1
```

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

a7ec1c
194b65

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
  = 0x7ffffffffff85ee0
- INT64_MAX = 0x7fffffffffffffff
- Sum of 2 CTx = 0xffffffffff0bdc0 (overflow)
  = -1000000_{10} = -0.01BTC
  - Difference between input & output interpreted as 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;
}
```

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;
18   static const unsigned int MAX_BLOCK_SIZE = 1000000;     | 18   static const unsigned int MAX_BLOCK_SIZE = 1000000;
19   static const int64 COIN = 100000000;                    | 19   static const int64 COIN = 100000000;
20   static const int64 CENT = 1000000;                      | 20   static const int64 CENT = 1000000;
                                                             | 21  +static const int64 MAX_MONEY = 21000000 * COIN;
21   static const int COINBASE_MATURITY = 100;               | 22   static const int COINBASE_MATURITY = 100;
22                                                           | 23
23   static const CBigNum bnProofOfWorkLimit(~uint256(0) >> 32); | 24   static const CBigNum bnProofOfWorkLimit(~uint256(0) >> 32);
@@ -471,10 +472,18 @@ class CTransaction
471          if (vin.empty() || vout.empty())                | 472          if (vin.empty() || vout.empty())
472              return error("CTransaction::CheckTransaction() : vin or vout empty"); | 473              return error("CTransaction::CheckTransaction() : vin or vout empty");
473                                                          | 474
474  -       // Check for negative values                    | 475  +       // Check for negative or overflow output values
                                                             | 476  +       int64 nValueOut = 0;
475          foreach(const CTxOut& txout, vout)              | 477          foreach(const CTxOut& txout, vout)
                                                             | 478  +       {
476              if (txout.nValue < 0)                       | 479              if (txout.nValue < 0)
477                  return error("CTransaction::CheckTransaction() : txout.nValue negative"); | 480                  return error("CTransaction::CheckTransaction() : txout.nValue negative");
                                                             | 481  +           if (txout.nValue > MAX_MONEY)
                                                             | 482  +               return error("CTransaction::CheckTransaction() : txout.nValue too high");
                                                             | 483  +           nValueOut += txout.nValue;
                                                             | 484  +           if (nValueOut > MAX_MONEY)
                                                             | 485  +               return error("CTransaction::CheckTransaction() : txout total too high");
                                                             | 486  +       }
478                                                          | 487
479          if (IsCoinBase())                               | 488          if (IsCoinBase())
480          {                                               | 489          {
```

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 attacker's 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.

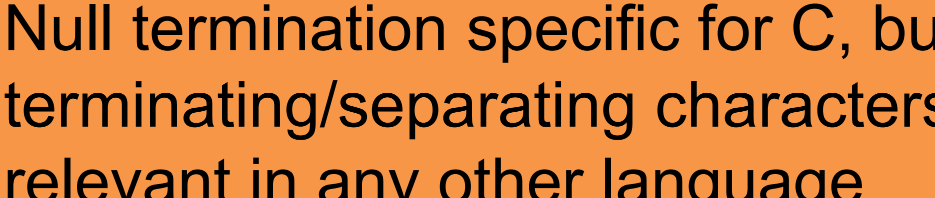
`num`
: Maximum number of characters to be copied from *source*.
size_t is an unsigned integral type.

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

first HeapHdr for all available memory

free memory

memory assigned by the kernel

SimpleHeap_alloc()

first HeapHdr allocated

usable memory

HeapHdr for remaining free memory

free memory

next

prev

allocation 1

usable memory

allocation 2

usable memory

HeapHdr for the remaining free memory

free memory

SimpleHeap_free()

allocation 1

usable memory

free 2

free memory

HeapHdr for the remaining free memory

free memory

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

Linked list between allocated blocks

Corrupted *allocation 2* data are later processed by unlink() function

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

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
     - vulnerable and safe functions for string manipulations
   - proper input checking
2. Protection by extensive testing (source code/binary/bytecode level)
   - automatic detection by static and dynamic checkers
   - code review, security testing
3. Protection by compiler (+ compiler flags)
   - runtime checks introduced by compiler (stack protection)
4. Protection by execution environment
   - DEP, ASLR, sandboxing, hardware isolation...

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
  - https://docs.microsoft.com/en-us/cpp/c-runtime-library/security-enhanced-versions-of-crt-functions
  - https://docs.microsoft.com/en-us/cpp/c-runtime-library/security-features-in-the-crt
  - 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, sscanf_s, swscanf_s, vfscanf_s, vfwscanf_s, vscanf_s, vwscanf_s, vsscanf_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

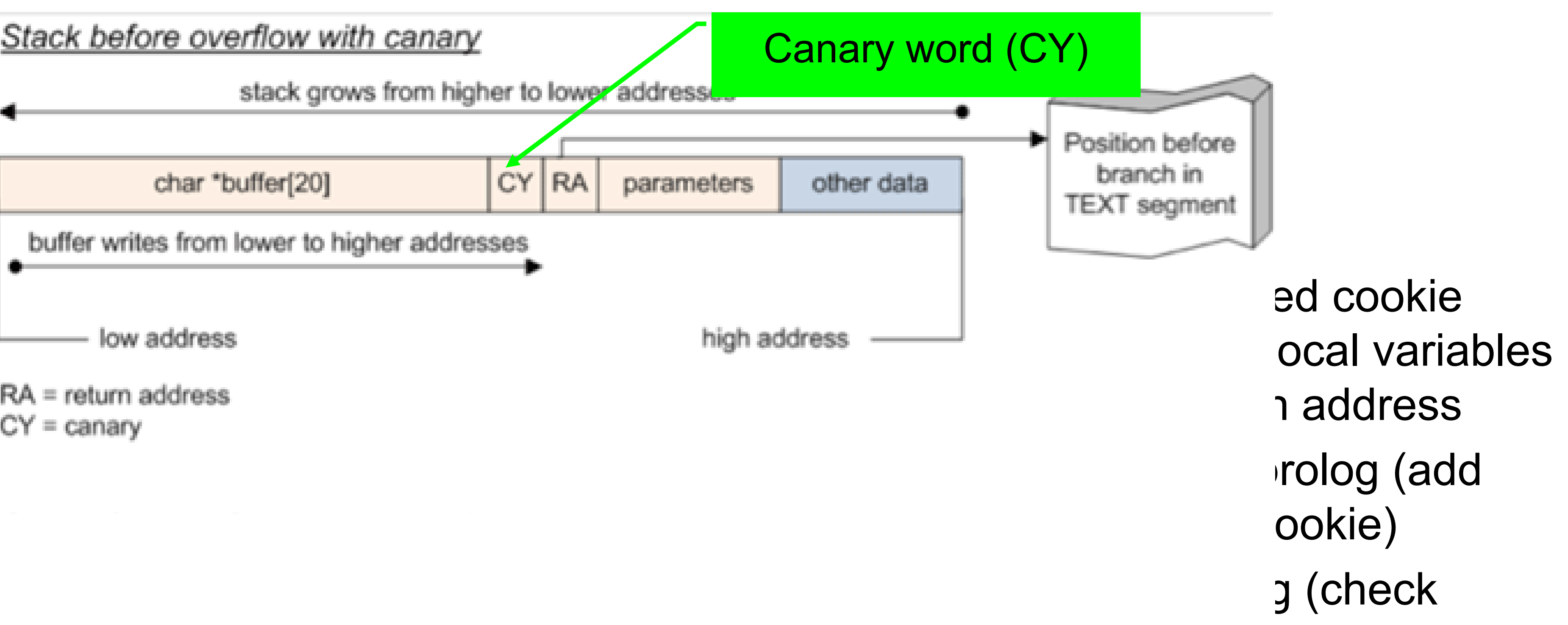
Secure C library – selected functions

- 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...

SOURCE CODE PROTECTIONS

COMPILER PROTECTIONS

PLATFORM PROTECTIONS



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

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

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 (...)

Automatic tools add vital protections, but are NOT replacement for secure defensive programming

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 -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 resu
35     call    __stack_chk_fail    ;                => call __stack
36
37 .L3:
38     leave                       ; clean-up stack
39     ret                         ; return
40     .cfi_endproc
```

```c
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 }
```

?

Can attacker bypass stack canary?

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

What attacker can do with changed return address?

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
- More in 5th lecture (Writing exploits)

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

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)

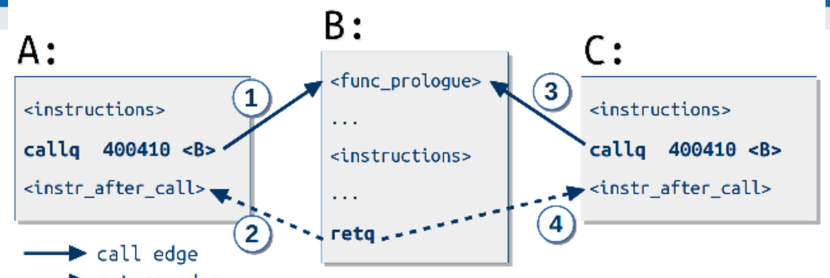
ASLR – impact on attacks

- 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

Can attacker execute desired unctionality without changing code?

Return-oriented programming (ROP)

- Return-into-library technique (Solar Designer, 1997)
  - method for bypassing DEP
  - no write of attacker’s code to stack (as is prevented by DEP)
  1. function return address replaced by pointer to standard library function
  2. library function arguments replaced according to attackers needs
  3. function return results in execution of library function and given arguments
  - Example: system call wrappers like system()
- Borrowed code chunks
  - Problem: 64-bit hardware introduced different calling convention
    - first arguments to function passed in CPU registers instead of via stack
  - attacker tries to find instruction sequences from any function that pop values from the stack into registers (automated search by ROPgadget)
  - necessary arguments are inserted into registers
  - return-into-library attack is then executed as before



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

More in 6th lecture (Return-oriented Programming)

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

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

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 attacker's 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

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 safer languages and safer 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: 2017 State of Application Security
  - https://www.sans.org/reading-room/whitepapers/application/2017-state-application-security-balancing-speed-risk-38100
  - Which applications are of main security concern?
  - What is expected time to deploy patch for critical security vulnerability?
  - How does your organization test applications for vulnerabilities?
  - Which language is the most common source of security risk?
- Previous years are also worth of reading
  - https://www.sans.org/reading-room/whitepapers/application/

Checkout

- The most interesting thing learned today?

Questions