

# Secure Programming Lecture 6: Memory Corruption IV (Countermeasures)

David Aspinall, Informatics @ Edinburgh

7th February 2017

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection

- Diversification

Secure programming

Summary

# Lab session **this afternoon**

The first Secure Programming Laboratory will be today!

***3pm-6pm in Forrest Hill labs 1.B31, 1.B32.***

Please **arrive on time**, there will be a short introduction in the first half hour of the lab (delivered in groups).

Recommended:

- ▶ find someone to work with (pairs rather than larger groups), for all the labs.
- ▶ preparation: study the lectures on overflows carefully, try out some examples.

If you can't come at 3pm because of a lecture clash, come at 4pm and speak to one of the demonstrators.

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection

- Diversification

Secure programming

Summary

# Memory corruption attacks

We've seen memory corruption attacks on the *stack*, on the *heap* and mentioned other locations.

Overflow vulnerabilities in code are caused by, for example:

- ▶ unchecked buffer boundaries
- ▶ out-by-one errors
- ▶ integer overflow
- ▶ type confusion errors

# Memory corruption countermeasures

Two basic programming-related countermeasures:

## 1. Treat the symptoms:

- ▶ special technologies in execution or compilation
- ▶ limit the damage that can be done by attacks
- ▶ **containment** and **curtailment**

## 2. Treat the cause:

- ▶ ensure that code does not contain vulnerabilities
- ▶ **secure programming** through code review, analysis tools

**Question.** Why might choice 2 be impossible?

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection

- Diversification

Secure programming

Summary

# Generic defences

Defensive technologies are not a real substitute for proper fixes, but:

- ▶ give *defence in depth* that can protect in case of new attacks, malware, regressions to vulnerable code
- ▶ sometimes code replacement is simply *prohibitively expensive* or *impossible* (e.g., non-upgradeable firmware)

**Question.** Can you give/find some examples of the latter?



# Defences against overflows

Several generic protection mechanisms have been invented to prevent overflow attacks and new ones are evolving.

These reduce the attackers chance of reliably exploiting a bug on the host system.

We will look at:

- ▶ Tamper detection in software
- ▶ Memory protection in OS and hardware
- ▶ Diversification methods

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection

- Diversification

Secure programming

Summary

# Canaries on the stack

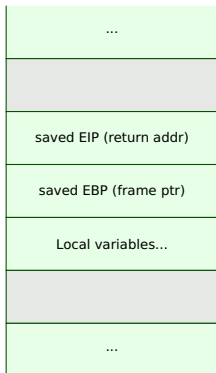
Each stack frame includes vulnerable location pointers which may be corrupted in a stack overflow attack.

Idea:

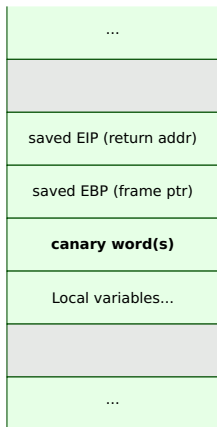
- ▶ wrap frame with protective layer, a “canary”
- ▶ canary sits below return address
- ▶ attacker overflows stack buffer to hit return address
  - ▶ necessarily overwrites canary
- ▶ generated code adds and checks canaries

Early proposal: *StackGuard* compiler.

# Stack without canaries



# Stack with canary



The "canary" is special data written into the stack to detect unexpected modifications. If a stack overflow or other corruption occurs, the canary may be altered. The compiler adds extra instructions to insert canaries and check their integrity.

**Question.** How might the mechanism be defeated?

**Question.** What should happen if an overflow is detected?

# GCC's Stack Smashing Protector

Consider this C program:

```
#include <stdio.h>
#include <string.h>

int fun1(char *arg) {
    char buffer[1024];
    strcpy(buffer, arg);
}

void main(int argc, char *argv[]) {
    fun1(argv[1]);
}
```

Let's compare the assembler compiled with `gcc -S -m32` and `gcc -S -m32 -fno-stack-protector`.

**main:** ; code without SSP: gcc -S -m32 -fno-stack-protector

```
    pushl    %ebp
    movl     %esp, %ebp
    andl     $-16, %esp      ; align stack to 16-byte
    subl     $16, %esp      ;
    movl     12(%ebp), %eax   ; eax = addr of argv
    addl     $4, %eax        ; eax = addr of argv[1]
    movl     (%eax), %eax    ; eax = contents of argv[1]
    movl     %eax, (%esp)    ; push it
    call     fun1           ;
    leave
    ret
```

**fun1:**

```
    pushl    %ebp           ; save old frame ptr
    movl     %esp, %ebp     ; set new frame ptr
    subl     $1048, %esp    ; allocate stack space
    movl     8(%ebp), %eax   ;
    movl     %eax, 4(%esp)   ; push arg (strcpy src)
    leal     -1032(%ebp), %eax
    movl     %eax, (%esp)    ; push buffer (strcpy dest)
    call     strcpy
    leave
    ret
```

```

fun1: ; code with SSP (main function stays the same)
        ; NB: GS register points to per-CPU thread storage
        pushl    %ebp
        movl     %esp, %ebp
        subl     $1064, %esp          ; use 16 bytes more this time
        movl     8(%ebp), %eax        ; fetch arg
        movl     %eax, -1052(%ebp)    ; >> keep a copy in our frame
        movl     %gs:20, %eax        ; >> set EAX=canary value
        movl     %eax, -12(%ebp)     ; >> store near return address
        xorl     %eax, %eax
        movl     -1052(%ebp), %eax    ; fetch local copy of arg
        movl     %eax, 4(%esp)        ; push it
        leal     -1036(%ebp), %eax    ;
        movl     %eax, (%esp)         ; push buffer
        call     strcpy
        movl     -12(%ebp), %edx      ; >> EDX=canary from stack
        xorl     %gs:20, %edx         ; >> has it changed?
        je       .L3
        call     __stack_chk_fail    ; if it has, we'll abort

.L3:
        leave
        ret

```



The stack protection spots an overflow with 1026 characters:

```
$ gcc -m32 overflow.c -o overflow.out
$ ./overflow.out xxxx
$ ./overflow.out `perl -e 'print "x"x1025'`
*** stack smashing detected ***: ./overflow.out terminated
Aborted (core dumped)
```

**Exercise.** Try this example for yourself, compiling with/without protection, and stepping through it using gdb. Draw the stack layout in each case. Make up some more complex examples and try them out.

# Security “arms race” and canaries

Attackers respond to new protection mechanisms by looking for vulnerabilities in those mechanisms (as well as new vulns).

For example:

- ▶ Attack code/probing discovers a constant canary
  - ▶ e.g., canary is 0x0af237ab6, so write that near return address
- ▶ Canary defence uses pseudorandom sequence
  - ▶ attacker learns sequence or discovers seed
- ▶ Canary defences uses *cryptographic* PRNG
  - ▶ attacker finds where value is stored
  - ▶ finds another exploit to copy it

# Stack canary effectiveness

- ▶ Doesn't protect against local variable overwriting
  - ▶ related mechanisms *reorder* local variables
- ▶ Other attacks work by overwriting parameters
  - ▶ aim to change where subsequent writes occur
  - ▶ overwrite return address, but don't return

Hardened heap implementations have also been developed

- ▶ glibc and Windows since XP SP2 have heap canaries
- ▶ but application specific heaps, HLLs not covered

# Better attacks, better detection

- ▶ Return-to-libc, and **return-oriented programming** (ROP)
  - ▶ state-of-the-art: use existing executable code
  - ▶ evades canaries, also defeats NX (see later)

A more powerful and defence mechanism is *Control-Flow Integrity*, which ensures that code execution follows a pre-determined call graph.

This can defend against ROP and similar attacks, depending on the accuracy and granularity of the enforcement.

# Outline

Announcement

Recap

Containment and curtailment

Tamper detection

**Memory mode protection**

Diversification

Secure programming

Summary

# Operating system separation (review)

**Isolation** different processes have different resources (address spaces, file systems, . . .)

**Sharing** resources are shared between processes, partial isolation. Sharing may be:

- ▶ all or nothing
- ▶ mediated with access controls
- ▶ mediated with *usage* controls (capabilities)

Concern: *granularity* of protection.

OSes have provided separation mechanisms since the early days of multi-user systems. For memory, direct support was added to the CPU and memory system hardware.

# Hardware memory protection mechanisms

Original mechanisms introduced to provide separation (mainly for safety) between different programs on multi-user systems:

- ▶ **Fences:** separate memory accesses between OS and user code (one boundary, one way protection).
- ▶ **Base and bounds registers:** enforce separation between several programs allowing access control on memory *ranges*.
- ▶ **Tagged architecture:** more fine-grained, tags on each memory location set access rights to stored word (R, RW, X). Supervisor mode instructions required to set tag. Not currently supported in modern architectures.

# Memory separation: segmentation & paging

**Segmentation** splits a program into named variable-sized logical pieces, (main,data,module,...). Programs use names and offsets; segment registers and an OS segment table for indexing.

**Paging** splits a program into fixed-sized pieces. These get mapped onto memory which is split into equal sized *page frames*.

Some OSes use a *flat memory model* without hardware-supported segmentation. The x86\_64 architecture and Linux work over a flat model.

**Exercise.** Investigate the pros and cons of each mechanism, particularly for security (consider possible attacks).



# Non-executable memory pages

CPUs have often included R, RW, X protection for memory pages.

- ▶ x86 series CPUs added page-level XD/NX in 2001-4

By enforcing non-executable regions, if the program keeps code and data separate, shellcode can be prevented from running when it's injected into data regions on the heap or stack.

- ▶ Compared with C, more tricky for certain languages/compilers/interpreters, which may manipulate executable code during runtime.

# Outline

Announcement

Recap

Containment and curtailment

Tamper detection

Memory mode protection

**Diversification**

Secure programming

Summary

# Address Space Layout Randomization (ASLR)

**Concept:** use *diversification* to make many versions of same program; thwarts general attacks that make assumptions about fixed structure.

**ASLR:** make it harder to find data or code locations, by randomising layout during load time. Breaks hard-coded static locations.

Implemented in Linux by the **PaX Team**.

**Effectiveness:** good, but doesn't remove main vulnerability and vulnerabilities in ASLR implementation become target of attack. Early implementations randomised by small amounts (e.g. 256 addresses), so attacker could use brute force to find the vulnerable locations. Such attacks may attract attention (since failures cause crashes).

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection

- Diversification

Secure programming

Summary

# Defensive programming: bounds checking

Defensive programming to avoid overflow requires **bounds checking**.

- ▶ Check **data lengths** before writing
- ▶ Check **array subscripts** are within limits
- ▶ Check **boundary conditions** to avoid OBO
- ▶ Constrain **size of inputs**
- ▶ Beware of **dangerous API calls** to risky code

# Responsibility for bounds checking

Like many security checks, this is a *shared responsibility*. It requires checking at each point, by the:

- ▶ programmer
- ▶ programming language, compiler
- ▶ OS
- ▶ hardware

**Exercise.** For each role, give an example of what could be done to check bounds.

# Bounds checks by programmer

```
int a[20], i;  
for (i=0, i<20; i++) {  
    a[i] = 0;  
    ...  
}
```

**Question.** How can this go wrong?

# Bounds checks by programmer

```
int a[20], i;  
for (i=0, i<20; i++) {  
    if (i<0) signal error;  
    if (i >= 20) signal error;  
    a[i] = 0;  
    ...  
}
```

- ▶ Checking every time seems inefficient
- ▶ Are both checks required?
- ▶ Tempting to skip...



# Bounds checks by programmer

```
int a[20], i, max;
...
for (i=0, i<max; i++) {
    if (i<0) signal error;
    if (i >= 20) signal error;
    a[i] = 0;
    ...
}
```

- ▶ If bound is computed, *both* checks essential
- ▶ Code reviews, programmer reasoning are *brittle*

# Safety from programming languages

Programming languages may provide memory safety and type safety, automatically for all programs:

**Memory safety** disallow reading/writing with arbitrary memory addressing.

Prevents overflow attacks from corrupting memory.

**Type safety** prevent storing arbitrary data into data values.

Makes it harder for attacker to inject data that will be executed as binary code, or cause crashes due to invalid representations.

# Safety from compilers and tools

To try to ensure safety with an unsafe language, we may use:

- ▶ a **safe compiler** to automatically generate checking code that checks bounds or types dynamically, *during execution*.
- ▶ a **verifying compiler** that checks *statically at compile time* that the code it produces is safe.
- ▶ **security testing tools** that generate inputs to programs to try to find security bugs.
- ▶ **program analysis tools** that ensure that input source code is free from certain vulnerabilities.

We'll look at these technologies in more detail later in the course.

# Safety from OS, libraries

- ▶ See lists of unsafe functions and explanations:
  - ▶ [Microsoft's recommendations](#)
  - ▶ CERT Secure C Coding
- ▶ Use a **code security scanning tool**
  - ▶ Have lists of dangerous API calls built in
  - ▶ Simple to find these in code, need deeper analysis to identify *certain* vulnerabilities
  - ▶ Examples: [RATS](#), [cppcheck](#), [SPLint](#), [Clang](#).
- ▶ Switch to using safe(r) library functions
  - ▶ The [Safe C Library](#) introduced in [VS 2005](#).
  - ▶ Bounds checking functions part of the latest C11 standard (appendix K, also [ISO/IEC TR 24731-1](#))
  - ▶ But has been [contentious](#), not clear how widely/quickly will be adopted.

# Outline

Announcement

Recap

Containment and curtailment

- Tamper detection

- Memory mode protection



- Diversification

Secure programming



Summary

# Review questions

## Overflow protection mechanisms

- ▶ Explain how StackGuard canaries prevent overflows. What attacks are they *not* effective against? 
- ▶ Describe how hardware-assisted memory protection can prevent the worst kinds of overflow attacks. In what cases may it be difficult to use? 
- ▶ Explain the strategy of program *diversification* and how it is achieved in ASLR.

## Avoiding overflow vulnerabilities

- ▶ Explain where bounds checking should be performed, especially to ensure “defence-in-depth”. 
- ▶ List some checks which a programmer or static analysis tool should do to prevent overflow vulnerabilities in released code. 

## Coming next

Lab session this afternoon!

Next lecture: looking at injections.