Secure Programming Lecture 5: Memory Corruption III (Countermeasures)

David Aspinall, Informatics @ Edinburgh

1st February 2018

Memory corruption recap

Buffer overflow is still one of the most common vulnerabilities being discovered and exploited in commodity software.

We've seen examples of **stack and heap buffer overflow** vulnerabilities due to copying without checking bounds.

In this lecture we'll see other memory corruption vulnerabilities and discuss countermeasures.

Buffer overflow risks have been known for over 30 years.
Is it still a problem? Try searching at `https://nvd.nist.gov` to see.

Outline

Other memory corruption errors

Copying data from one place to another isn't the only source of memory corruption.

Other mistakes can be made by errors with

- out-by-one errors
- overflowing data values
- pointer arithmetic

Exercise. Find and explain an example of a pointer arithmetic bug leading to a code vulnerability.

Outline

Out-by-one errors

- Mistaking the size of array

```
for (i=0; i<=sizeof(dest); i++)
  dest[i]=src[i];
```

- Forgetting to account for string terminator in C

```
if (strlen(user) > sizeof(buf))
   die("user string too long\n");
strcpy(buf, user);
```

These are typical programming errors.

They *may* cause exploitable memory corruption, depending on the rest of the application code.

Integer overflow

Integer overflow (wrap-around) can cause memory corruption.

Worrying case: bounds calculated from user inputs.

```
char *make_table(int width, int height, char* defaultrow) {
    char *buf;
    int i;
    int n = width * height;
    buf = (char*)malloc(n);
    int i;
    if (!buf)
        return NULL;
    for (i=0; i<height; i++)
        memcpy(&buf[i*width], defaultrow, width);
}
```

Exercise. Show that with carefully chosen width and height, it’s possible to perform a massive overflow.

Outline

Typing discpline

Type safety

A programming language, analysis tool or runtime is said to enforce **type safety** if it has a clearly specified typing discipline for data values and it ensures that data values (representations) for types stay within the domain of those types during program execution.

C is not type safe!

C has overly flexible typing:

- **implicit type conversions**, inserted automatically by the compiler, often for convenience of arithmetic combining differently sized primitives.
- **explicit type casts**, where the programmer writes `foo = (sometype) bar;`
  A value in one type is treated as a value of another type. For pointers, there is no effect: the pointed-to values are not altered.

Numeric conversions may perform *sign extension* or *truncation*.

Some conversions are implementation defined (i.e., are not pinned down by the language, so vary depending on the compiler, platform, etc).

Signed integer comparison vulnerability

```
int read_user_data(int socketfd) {
    int length;
    char buffer[1024];
    length = get_user_length(socketfd);

    if (length>1024) {
       error("Input size too large\n");
       return -1;
    }
    if (recv(socketfd, buffer, length)<0) {
       error("Read format error\n");
       return -1;
    }
    return 0; // success
}
```

- Here, a negative `length` defeats the size check...
- but recv accepts a `size_t` type, which is *unsigned*
- a negative value becomes a large positive one
- ...and `recv()` overflows `buffer`.

Outline

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?

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) so defences must be put elsewhere.

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 attacker's 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

Canaries on the stack

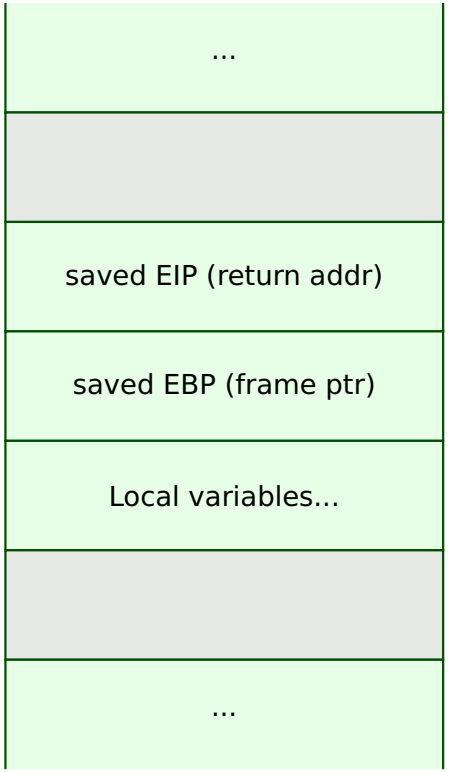
Each stack frame includes vulnerable location pointers which may be corrupted in a stack overflow attack.

Idea:

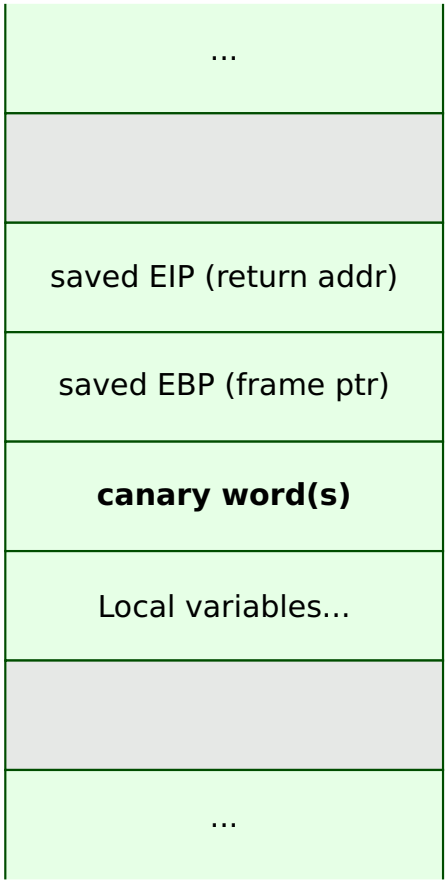
- wrap frame with protective layer, a “canary”
- canary sits below return address
- attacker overflows stack buffer to hit return address
  - necessarily overwrites canary
- generated code adds and checks canaries

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

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
- Data execution prevention: attempt to execute causes page-fault

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.

Apart from C, this may be tricky to use with certain languages/compilers/interpreters that manipulate executable code during runtime.

Outline

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

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

In a sense this is a *shared responsibility*: we may put trust in the tool-chain or each layer of the running platform to implement checks or ensure they are not needed.

Responsibility for bounds checking

Checks or guarantees could be given by the:

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

(we'll consider programmer checks now, others later).

Exercise. For each role, give an example of what could be done to check bounds and what might go wrong if a check isn't done.

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*

Outline

Memory corruption attacks and defences

We've seen memory corruption attacks *on the heap*, *on the stack* and elsewhere.

Vulnerabilities in code are caused by:

- unchecked buffer boundaries
- pointer arithmetic errors
- out-by-one errors
- value overflow, type confusion

Countermeasures are generic defences or secure programming:

- Detect and abort: canaries
- Diversification: address randomisation (ASLR)
- Execution prevention (NX), Control Flow Integrity
- Programming: bounds checking, library functions

Review questions

Memory corruption vulnerabilities

- Explain type confusion errors, giving an example.

Protection mechanisms

- Explain how StackGuard canaries prevent overflows. What attacks are they *not* effective against?
- How does hardware-assisted memory protection work and when may it be difficult to use?
- Explain the strategy of program *diversification* and how it is achieved in ASLR.

Avoiding overflow vulnerabilities

- Describe where bounds checking should be used to ensure “defence-in-depth”.

References and credits

- Some of the examples were adapted from *The Art of Software Security Assessment*.