Secure Programming Lecture 3:
Memory Corruption I (Stack Overflows)

David Aspinall, Informatics @ Edinburgh

24th January 2017

Roadmap: Security in the software lifecycle

Security flaws (and other problems) may be introduced at different stages in the Secure Software Development Life Cycle (SSDLC). Key stages are:

1. Design
2. Implementation
3. Deployment

We focus mainly on coding flaws.

Roadmap: From vulnerabilities to fixes

This course emphasises removing vulnerabilities in software rather than crafting exploits to break code. But insight into exploits is needed to understand the reason for vulnerabilities. It also helps to how and why defensive programming practices and tools work.

Roadmap: Kinds of vulnerability

Software vulnerabilities fall into categories, for example:

- memory corruption errors
- injection
- broken authentication
- bad cryptography

and others. We’ll look at vulnerability taxonomies later.

Memory corruption vulnerabilities

This lecture begins our look at code vulnerabilities, starting with memory corruption.

Memory corruption vulnerabilities let the attacker cause the program to write to areas of memory (or write certain values) that the programmer did not intend. In the worst cases, these can lead to arbitrary command execution under the attacker’s control.

We examine vulnerabilities, exploits, defences and repair.

Reasons for memory corruption

Memory corruption vulnerabilities arise from possible:

- buffer overflows, in different places
- stack overflows
- heap overflows
- other programming mistakes
- out-by-one errors
- type confusion errors
The nuts and bolts

We begin with classic stack overflows. Some of you may have seen examples of this already. The idea is to bring everyone to the same point, so we give some “instant” background.

If you don’t know these details already, this lecture will indicate where you should study more.

Make sure you understand Lectures 3, 4 and 5 before the first lab exercise. If you have only studied high-level programming languages before you should spend time on this or you will find the lab difficult.

Programming in C or assembler

▶ Low-level programs manipulate memory directly
▶ Advantage: efficient, precise
▶ Disadvantage: easy to violate data abstractions
  ▶ arbitrary access to memory
  ▶ pointers and pointer arithmetic
  ▶ mistakes violate memory safety

Memory safety

A programming language or analysis tool is said to enforce memory safety if it ensures that reads and writes stay within clearly defined memory areas, belonging to different parts of the program. Memory areas are often delineated with types and a typing discipline.

Von Neumann programming model

▶ Von Neumann model:
  ▶ code and data are the same stuff
▶ Von Neumann architecture
  ▶ implements this in hardware
  ▶ helped revolution in Computing 1950s-1970s
▶ But has drawbacks:
  ▶ data path and control path overloaded (bottleneck)
  ▶ code/data abstraction blurred
  ▶ self-modifying code not always safe. . .

Close to the metal

Question. What are the trusted bits of code in this picture? In what way do we trust them?

Further from the metal

Question. What are the trusted bits of code in this picture? In what way do we trust them?

Processes and memory

A process is a running program managed by the operating system.

Processes are organised into several memory areas:

1. **Code** where the compiled program (or shared libraries) reside.
2. **Data** where non-local program variables are stored. This contains global or static variables and the program heap for dynamically allocated data.
3. **Stack** which records dynamically allocated data for each of the currently executing functions/methods. This includes local variables, the current object reference and the return address.

The OS (with the CPU, language runtime) can provide varying amounts of protection between these areas.
Instant C programming

- You should know Java. C uses a similar syntax.
- It has no objects but
  - pointers to memory locations (&val, *ptr)
  - arbitrary-length strings, terminated with ASCII NULL
  - fixed-size structs for records of values
  - explicit dynamic allocation with malloc()
- It has no exceptions but
  - function return code conventions
- Is generally more relaxed
  - about type errors
  - uninitialised variables
- But modern compilers give strong warnings
  - these days, errors by default
  - can instrument C code with debug/defence code

```
#include <stdio.h>
void main(int argc, char *argv[]) {
    int c;
    printf("Number of arguments passed: %d\n", argc);
    for (c = 0 ; c < argc ; c++) {
        printf("Argument %d is: %s\n", c, argv[c]);
    }
}
```

$ gcc showargs.c -o showargs
$ ./showargs this is my test
Number of arguments passed: 5
Argument 0 is: ./showargs
Argument 1 is: this
Argument 2 is: is
Argument 3 is: my
Argument 4 is: test

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

typedef struct list { int hd; struct list *tl; } list_t;

void printlist(list_t *l) {
    while (l != NULL) {
        printf("%i\n",l->hd); l=l->tl;
    }
}

int main(int argc, char *argv[]) {
    int c; list_t *cell = NULL;
    for (c = argc-1; c > 0; c--) {
        list_t *newcell = malloc(sizeof(list_t));
        (*newcell).hd = (int)(strlen(argv[c]));
        newcell->tl = cell;
        cell = newcell;
    }
    if (cell != NULL) printlist(cell);
}
```

$ gcc structeg.c -o structeg
$ ./structeg this is my different test
4
2
2
9
4

Exercise. If you haven’t programmed C before, try these examples and some others simple programs from a textbook. Write a program to reverse its list of argument words.

Instant assembler programming

- x86: hundreds of instructions! But in families:
  - Data movement: MOV ...
  - Arithmetic: ADD, FDIV, IDIV, MUL, ...
  - Logic: AND, OR, XOR, ...
  - Control: JMP, CALL, LEAVE, RET, ...
- General registers are split into pieces:
  - 32 bits : EAX (extended A)
  - 16 bits : AX
  - 8 bits : AH AL (high and low bytes of A)
- Others are pointers to segments, index offsets
  - ESP: stack pointer
  - EBP: base pointer (aka frame pointer)
  - ESI, EDI: source, destination index register

(We’ll stick to x86 32-bit instructions in this course, 64-bit is slightly different).
Here is a file `movc.c`:

```c
int value;
int *ptr;

void main() {
    value = 7;
    ptr = &value;
    *ptr = value * 13;
}
```

Compile this to assembly code with:

```bash
$ gcc showargs.c -S -m32 movc.c
```

This produces a file `movc.s` shown next.

```asm
.data
value:
.long 2
ptr:
.long 2
.text
    movl $7, %eax ; set EAX to 7
    movw %ax, value ; value is now 7
    movl $value, ptr ; set ptr = address of value
    movl ptr, %ecx ; ECX to same
    movl value, %edx ; EDX = 7
    movl %edx, %eax ; EAX = 7
    addl %eax, %eax ; EAX = 14 (2*7)
    addl %edx, %eax ; EDX = 7 + 14 = 21
    sall $2, %eax ; EAX = 21 * 4 = 84 (12 * 7)
    addl %edx, %eax ; EAX = 7 + 84 = 91 (13 * 7)
    movl %eax, (%ecx) ; set value = 91
```

Exercise. If you haven’t looked at assembly programs before, compile some small C programs and try to understand the compiled assembler, at least roughly.

Stack overflow: high level view

- The malicious argument overwrites all of the space allocated for the buffer, all the way to the return address location.
- The return address is altered to point back into the stack, somewhere before the attack code.
- Typically, the attack code executes a shell.

How the stack works

- Recall Abstract Data Type (encapsulation) principles:
  - access to data possible only by ADT operations
  - only data built via operations can be represented
- Recall the stack Abstract Data Type, a first-in first-out queue:
  - `push(X)`: add an element X to the top
  - `pop()`: remove and return the top element

How the stack works

The **program stack** (aka function stack, runtime stack) holds stack frames (aka activation records) for each function that is invoked.

- Very common mechanism for high-level language implementation
- So has special CPU support
  - `stack pointer registers`: on x86, ESP
  - `frame pointer registers`: on x64, EBP
  - push and pop machine instructions
- Exact mechanisms vary by CPU, OS, language, compiler, compiler flags.

NB: in pictures, some authors draw stacks growing upwards on the page! I will draw stacks that grow downwards.

Stack overflow attacks were first carefully explained in Smashing the stack for fun and profit, a paper written by Aleph One for the hacker’s magazine Phrack, issue 49, in 1996.

- Stack overflows are mainly relevant for C, C++ and other unsafe languages with raw memory access (e.g., pointers and pointer arithmetic).
- Languages with built-in **memory safety** such as Java, C#, Python, etc, are immune to the worst attacks — providing their language runtimes and native libraries have no exploitable flaws.
How the stack works

Stack
↑ high addresses
...
Data
↓ low addresses
Code
Memory

Stack usage with function calls

void fun1(char arg1, int arg2) {
  char *buffer[5];
  int i;
  ...
}

fun1 has two arguments arg1 and arg2.
- Actual parameters may be passed to the function body on the stack or in registers; the precise mechanism is called the calling convention.

fun1 has two local variables buffer and i.
- Local variables are allocated space on the stack.

A frame pointer may be used to help locate arguments and local variables.

Frame pointers

frame for main() { ...
  arg2
  arg1
  return address
  old frame ptr
  i
  buffer
  ...
}
frame for fun1() { ...
  return address
  i
  buffer
  ...
}

Assembly code for function calls

Let’s see assembly code produced by gcc compiling C programs on Linux (32 bit x86), using gcc -S.

int main() {
  return 0;
}
produces:

main:
pushl %ebp ; save EBP, the old frame pointer
movl %esp, %ebp ; the new frame pointer for body of main()

void fun1(char arg1, int arg2) {
  char *buffer[5];
  int i;
  +buffer[0] = (char)i;
}
void main() {
  fun1('a',77);
}
fun1:
    pushl %ebp ; save previous frame pointer
    movl %esp, %ebp ; set new frame pointer
    subl $36, %esp ; allocate enough space for locals
    movl -24(%ebp), %eax ; EAX = address of buffer[0]
    movl -4(%ebp), %edx ; EDX = i
    movb %dl, (%eax) ; Set buffer[0] to be low byte of i
    leave ; drop frame
    ret ; return

main:
    pushl %ebp ; save previous frame pointer
    movl %esp, %ebp ; set new frame pointer
    subl $8, %esp ; allocate space for fun1 parameters
    movl $77, 4(%esp) ; store arg2
    movl $97, (%esp) ; store arg1 (ASCII 'a')
    call fun1 ; invoke fun1
    leave ; drop frame
    ret ; return

Exercise. Draw the detailed layout of the stack when the frame for 'fun1()' is active.

Review questions

Program execution

- Explain the points of trust that exist when a Linux user runs a program by executing a binary file.

Buffer overflows

- How do they arise?
- In what sense are some languages considered immune from buffer overflow attacks?

Runtime stack basics

- Describe how function parameters and local variables are allocated on the runtime stack.

Next time

We’ll continue looking at the detail of stack and buffer overflow exploits.