Secure Programming Lecture 5: Memory Corruption III (Heap and other attacks)

David Aspinall, Informatics @ Edinburgh

26th January 2016

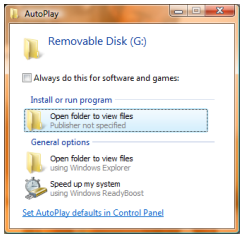
SQL Slammer Worm (2003)

SECURITY Feb 3, 2003 7:00 AM

Slammer Was Fastest Spreading Worm Yet

- overflow in MS-SQL server, pure stack overflow
- 100,000 machines affected. Shutdown ATMs, 911 emergency lines in Seattle.
- Extreme rapid spread: 8 seconds doubling time
- Small size: one UDP packet 376 bytes.
- Removed by reboot; only action was re-transmit.
- Still fastest ever, 10 years later.

Conficker (2008)



- Windows Server; originally used counting error then stack overflow. Newer variants used additional infection mechanisms, e.g. USB autoplay spoof.
- Around 10 million machines infected.
- Infected machines join botnet, wait for C&C
- MS $250k reward for information still unclaimed

Memory corruption

Buffer overflow is still one of the most common vulnerabilities being discovered and exploited in commodity software.
We've seen examples of **stack overflow** exploits based on buffer copying without checking bounds.
In this lecture we'll see **heap overflow** exploits, and explain some other causes of memory corruption.

Heap overflows: overview

The **heap** is the region of memory that a program uses for dynamically allocated data.
The runtime or operating system provides *memory management* for the heap.
With *explicit* memory management, the programmer uses library functions to allocate and deallocate regions of memory.

Memory allocation in C

`malloc(size)` tries to allocate a space of `size` bytes.

- It returns a pointer to the allocated region
- . . . of type **`void*`** which the programmer can *cast* to the desired pointer type
- or it **fails** and returns a NULL pointer
- The memory is **uninitialised** so should be written before being read from

Question. Which points above contribute to *unsafe* behaviour in C?

Memory allocation in C

`calloc(size)` behaves like `malloc(size)` but it also initialises the memory, clearing it to zeroes.

Question. Suppose we allocate a string buffer, and immediately assign the empty string to it.
What security reason may there be to prefer ‘calloc()‘ over ‘malloc()‘?

Memory allocation in C

`free(ptr)` frees the previously allocated space at `ptr`.

- No return value (`void`)
- If it fails (`ptr` a non-allocated value), what happens?
  - if `ptr` is `NULL`, nothing
  - “undefined” otherwise,
  - program may abort, or might carry on and let bad things happen
- What happens if `ptr` is dereferenced after being freed?
  - depends on behaviour of allocator

Question. Suppose we accidently call ‘free(ptr)‘ before the final dereference of ‘ptr() but before another call to ‘malloc()‘. Is that safe?

Simple heap variable attack

Without memory safety, heap-allocated variables may overflow from one to another.

```
char *user = (char *)malloc(sizeof(char)*8);
char *adminuser = (char *)malloc(sizeof(char)*8);

strcpy(adminuser, "root");

if (argc > 1)
  strcpy(user, argv[1]);
else
  strcpy(user, "guest");

/* Now we'll do ordinary operations as "user" and
   create sensitive system files as "adminuser" */
```

- Is it possible to overflow `user` and change `adminuser` ?

Simple heap variable attack

Problem: how do we know where the allocations will be made?

- Heap allocator is free to allocate anywhere, not necessarily in adjacent memory

Let’s investigate what happens on Linux x86, glibc.

Simple heap variable attack

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

void main(int argc, char *argv[]) {

  char *user = (char *)malloc(sizeof(char)*8);
  char *adminuser = (char *)malloc(sizeof(char)*8);

  strcpy(adminuser, "root");

  if (argc > 1)
    strcpy(user, argv[1]);
  else
    strcpy(user, "guest");

  printf("User is at %p, contains: %s\n", user, user);
  printf("Admin user is at %p, contains: %s\n", adminuser, adminuser);
}
```

```
$ gcc useradminuser.c -o useradminuser.out
$ ./useradminuser.out
User is at 0x9504008, contains: guest
Admin user is at 0x9504018, contains: root

$ ./useradminuser.out
User is at 0x9483008, contains: guest
Admin user is at 0x9483018, contains: root

$ ./useradminuser.out frank
User is at 0x8654008, contains: frank
Admin user is at 0x8654018, contains: root
```

- Buffers not adjacent, there’s some extra space
- Addresses not identical each run (next lecture. . .)
- But admin user *is* stored higher in memory!

Let's try overflowing....

```
$ ./useradminuser.out frank........david
User is at 0x9405008, contains: frank........david
Admin user is at 0x9405018, contains: id
```

Count more carefully:

```
$ ./useradminuser.out frank56789ABCDEFdavid
User is at 0x9f0b008, contains: frank56789ABCDEFdavid
Admin user is at 0x9f0b018, contains: david
```

Whoa!

Question. Can you think of a way to prevent this attack?

Remarks about heap variable attack

- same kind of attack is possible for (mutable) **global variables**, which are allocated statically in another memory segment
- this is an **application-specific** attack, need to find security-critical path near overflowed variable
- need to be lucky: overwriting intervening memory might cause crashes later, before the program gets to use the intentionally corrupted data

Is there a more generic attack for the heap?

Heap allocator implementation

A common heap implementation is to use blocks laid out contiguously in memory, with a *free list* intermingled. Heap blocks have *headers* which give information such as:

- size of previous block
- size of this block
- flags, e.g., *in-use* flag
- if not in use, pointers to next/previous free block

The doubly-linked free list makes finding spare memory fast for the `malloc()` operation.

Heap allocator implementation

```
typedef struct mallocblock {
    struct mallocblock *next;
    struct mallocblock *prev;
    int prevsize;
    int thissize;
    int freeflag;
    // malloc space follows the header
} mallocblock_t;
```

- If `freeflag` is non-zero, the block is in the freelist
- Allocator will split blocks and coalesce them again

General heap overflow attack

Rough idea:

- Coalescing blocks unlinks them from the free list
- Attacker makes `unlink()` do an arbitrary write!
  - uses overflow to set next and previous
  - and set flags to indicate free
  - `unlink()` then performs write

Unlinking operation

```
void unlink(mallocblock_t *element) {
  mallocblock_t *mynext = element->next;
  mallocblock_t *myprev = element->prev;

  mynext->prev = myprev;
  myprev->next = mynext;
}
```

- performs two (related) word writes
  - `mynext->prev=*mynext+2`, `myprev->next=*myprev`
- attacker arranges at least one of these to be useful

Exercise. Check you understand this: draw a picture of a doubly linked list and explain how the attacker can make an arbitrary write.

Writing to arbitrary locations

What locations might the attacker choose?

- *Global Offset Table* (GOT) used to link ELF-format binaries. Allows arbitrary locations to be called instead of a library call.
- *Exit handlers* used in Unix for return from `main()`.
- Lock pointers or exception handlers stored in the Windows *Process Environment Block* (PEB)
- Application-level function pointers (e.g. C++ virtual member tables).

The details are intricate, but library exploits and tookits are available (e.g., Metasploit).

Heap spraying and browser exploits

Apart from operating system (C code) memory management, other application runtimes provide memory allocation features, which may be accessible to an attacker.

A particular case is in **browser-based exploits** which have made use of heaps for managed runtimes such as **JavaScript**, **VBScript**, **Flash**, **HTML5**.

Writing shell code to predictable heap locations is sometimes called **heap spraying**. This is simple in concept: string variables manipulated in scripts are allocated in a heap.

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);
```

Typical programming errors, may cause exploitable memory corruption (overflow by one position), depending on the application.

Integer overflow

Integer overflow (wrap-around) can cause memory corruption errors.
Typical case: bounds are calculated based on user inputs.

```
char *make_table(int width, int height, char* defaultrow) {
    char *buf;
    int n = width * height;
    buf = (char*)malloc(n);
    if (!buf)
        return NULL;
    for (i=i; 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.

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`.

Memory corruption attacks

We've seen memory corruption attacks *on the heap*, *on the stack* and elsewhere.
Overflow vulnerabilities in code are caused at least by:

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

Review questions

Heap overflows

- Explain the API functions used to interface to heap allocation in C. Give two examples of risky behaviour.
- Show how overflowing one heap-allocated variable can corrupt a second.
- Sketch how a heap overflow attack can exploit memory allocation routines to write to locations controlled by an attacker.

Other vulnerabilities

- Explain type confusion errors, giving an example.

Coming next

Next time, we'll look at **countermeasures** to overflows, including **protection mechanisms** and **secure programming**.

References and credits

- The Conficker autoplay image is from The Register.
- See Sophos article about Slammer, 10 years on.
- Heap overflows were first described by Solar Designer. A longer article is in *Phrack 57*.
- Some of the examples were adapted from *The Art of Software Security Assessment*.