Software Security Course

Memory corruption and other memory-related vulnerabilities

Dimitrios A. Glynos
{ daglyn at unipi.gr }

Department of Informatics
University of Piraeus

Part I

Memory corruption bugs

Memory corruption bugs

Present in software developed in programming languages that

- allow direct memory references (Assembly, C, C++, ...)
  - `*p = value;`
- do not perform out-of-bounds checks in array-type operations
  - `a[++i] = 4;`

Memory corruption bugs

Also affect software that is based on components that are vulnerable to memory corruption bugs. For example:

- Java does not allow direct memory references and raises a runtime exception on array index errors
  - However, the JVM is vulnerable to memory corruption bugs
  - The JDK native libraries are vulnerable to memory corruption bugs
  - Executing untrusted bytecode on a JVM could trigger such bugs (e.g. browser applets)

Memory corruption bugs are everywhere

- Operating System Kernels
- System libraries
- System tools
- Software for embedded systems (routers, TVs, mobile devices, ...)
- Internet services
- Native libraries
- Web browsers
- Software supporting SCADA systems
- ...

Impact of memory corruption bugs

- Program continues working with modified in-memory data
- Runtime change in business logic
- Denial of Service (program crash)
- Code execution
- Security control bypass (runtime patch)
- Disclosure of sensitive information

Common memory corruption bugs

- Buffer overflows
- Off-by-one errors
- Untrusted pointer dereference bugs
    - write **something** *somewhere* type of errors
- Type Confusion[1] bugs
- Format string bugs

[1]for reasons of brevity we will not be exploring this type of bugs

Buffer overflows

- Data are written past the end of a buffer corrupting the contents of adjacent memory locations.
- Example:

```
/* attacker controls all img.hdr data */
memcpy(dest, &(img.hdr), img.hdr.size);
```

- What if an attacker makes `img.hdr.size` larger than the size of the `dest` buffer?
- Common types
    - Stack buffer overflows (local variables etc.)
    - Heap buffer overflows (dynamically allocated variables / objects)
- Other types
    - BSS buffer overflows (global writable variables)
- Proposal: perform bounds checking operations / limit the copy operation to the size of the destination buffer.

Buffer overflows

- Commonly due to the use of *unsafe* libc functions: gets(3), scanf(3), sprintf(3), strcpy(3), strcat(3), ...
  - These functions stop copying data when they find a NULL byte in the source buffer (or EOF for gets(3)), regardless of the size of the destination buffer

```
char argument[100];
strcpy(argument, argv[2]);
```

- *Safe* alternatives exist: fgets(3), snprintf(3), strncpy(3), strncat(3), ...
  - they allow programmers to specify the maximum number of bytes to be written to the destination buffer

Buffer overflows

Attackers can cause a buffer overflow when they control at least one of the below parameters

- the size of the data being copied
- the location of the input buffer[2]
- the data being copied (and software expects certain terminator symbol in data)
- the size of the output buffer
- the location of the output buffer

[2]this, on its own, may lead to information leaks

Exploiting a buffer overflow on the stack

- Function Call Stack

```
high address  ^   [ Function Parameters ]
              |   [    Return Address   ]
              |   [ Saved Frame Pointer ]
 low address  |   [   Local Variables   ]
```

- An overflow in a local variable may overwrite the function’s return address
- The return address specifies the instruction that will be executed once the function exits
- An attacker can exploit the overflow to redirect the program flow to:
  - code supplied as part of the overflow
  - other code within the memory space of the process
- Note: x86_64 and ARM64 use by default registers to pass function parameters

Exploiting a buffer overflow on the heap

- Overwrite heap metadata to cause an arbitrary memory write
- Overwrite a function pointer
- ...
- For more information see Phrack articles on heap exploitation subjects

Terminology

- **Shellcode**: instructions that are part of an exploit and typically provide the attacker with a shell on the target system
- **Nopsled**: NOP (No operation / Dummy) instructions used to drive the program flow towards the shellcode when its exact location in memory will not be known in advance

Heap Buffer Overflow Example

DEMO with Heap Playground

Off-by-one errors

- Example:

```
void func(...)
{
    char buf[255];
    char data[255];
    ...
    for (i = 0; i <= sizeof(buf); i++)
        buf[i] = data[i];
    ...
}
```

- Very popular bugs due to C semantics or API semantics (e.g. `strncpy` does not terminate the output string with NULL, copies up to *n* bytes)
- Proposal: Do not copy more bytes than the size of the destination buffer.

Exploiting off-by-one errors

One byte overwrite of saved frame pointer

- An off-by-one overflow to the variable right next to the saved frame pointer will change one byte in the saved frame pointer address
- This address may point to a fake stack frame within the attacker provided data
- At the epilogue of the vulnerable function: ESP = EBP, EBP = modified frame pointer
- At the epilogue of the caller: ESP = modified frame pointer, but the `ret` instruction will use an address found in the fake stack frame!

Non-executable pages

- The W ^ X philosophy
  - Writable pages should not be executable
- Modern CPUs allow pages to be marked as non-executable
  - AMD NX bit
  - Intel XD bit
- An executable file or a library can describe which sections are to be loaded at non-executable pages
- W ^ X can also be emulated in software
  - use of CS register to limit the executable part of a segment
  - ExecShield project
- Proactive security feature: Can stop attackers from executing code on the stack etc.

Stack protection

- Making the stack non-executable
- Reordering variables to protect function pointers
  - Function pointers are placed in lower memory addresses to protect them from overflows of arrays or other variables
- Canaries
  - A random value (canary) is placed right after the local arguments of a function
  - If the function exits and the canary has changed, it is a sign of an overflow and the application immediately exits
  - gcc -fstack-protector-all
  - Visual Studio /GS compiler option

Heap protection

- Making the heap non-executable
- Performing sanity checks on heap metadata during memory allocation / deallocation
- Canaries
  - Guard values to protect heap metadata
  - They are checked only during allocation / deallocation calls

Return oriented programming

- If the address of some part of the program code (executable / library / other code) can be guessed, then an attacker can borrow (read: jump to) that code to execute commands (or bypass memory protections)
- For example, if the address of `system()` can be guessed then the attacker can perform a “return to libc” attack, by effectively calling `system` with the right arguments
- Return oriented programming (ROP): next address to execute is popped off the stack due to a `ret` instruction

Return oriented programming

- Alternatively, by using snippets of borrowed code (called 'gadgets') an attacker can chain an exploit that:
    - allocates a new executable memory page
    - writes shellcode on that page
    - executes the shellcode
- Chaining techniques:
    - providing the program code with addresses of gadgets (e.g. through a buffer overflow on the stack)
        - each gadget executes and returns (so that the next return address will be used)
    - instead of return statements, other techniques (jumps etc.) can also be used

Address space layout randomisation (ASLR)

- Proactive security feature
- The OS loads the stack, heap, mmap-ed pages, program text/data and libraries to random memory locations each time an application is executed
  - Thus code and data are no longer accessible at static addresses
- Requires position independent code (PIC) for executable segments
- Executables that load their code segment to random addresses are called PIE (Position Independent Executables)
- If all pieces of a process are loaded at random addresses, ROP / ret-to-libc attacks can be thwarted (provided that the attacker cannot write data to executable pages)
- Brute force attacks are sometimes possible
  - Forked processes share the same memory layout

Untrusted pointer dereference bugs

- `a[attacker_controlled] = value;`
- `*(a + attacker_controlled) = value;`
- Opera CVE-2011-1824
  - `mov DWORD PTR [ebx+edx*1], eax`
- The attacker controls
  - the memory position where data will be written
  - and in some cases the data (and length) as well
- An arbitrary memory write operation
- Example exploitation: overwrite a function pointer!
- Proposal: perform input sanitization before using untrusted input in pointer arithmetic

Format string bugs

- Sometimes programmers allow user input to enter format strings

```
strcpy(fmt_buf, "user id: %i user: ");
strcat(fmt_buf, username);
sprintf(formatted_str, fmt_buf, id);
```

- Or allow users full control of format strings

```
printf(username);
```

- If an attacker places a %s specifier, the next address in the stack will be considered as pointing to a string and all contents there (leading to a NULL) will be dumped as part of the call.
- %n specifier: *The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.*
  - Each %n format specifier supplied by an attacker will write to an address found in the stack which can be controlled by the attacker!

Format string bugs

- 32-bit exploitation payload[3] for `printf(buf)` with `buf` on stack:

```
"\x94\x90\x04\x08" // GOT[free]'s address
"\x96\x90\x04\x08" //
"\x98\x90\x04\x08" // jumpcode address
"%.37004u"       // complete to 0x9098 (0x9098-3*4)
"%8$hn"          // write 0x9098 to eighth param. (0x8049094)
"%.30572u"       // complete to 0x10804 (0x10804-0x9098)
"%9$hn"          // write 0x0804 to nineth param. (0x8049096)
"%.47956u"       // complete to 0x1c358 (0x1c358-0x10804)
"%10$hn" // write code (pop eax - ret) @10th param. (0x8049098)
```

- Proactive protections: `gcc -Wformat=2`, VS2005 has %n disabled
- Proposal: Never allow user-controlled data to enter format strings. Always supply a constant format string argument to calls allowing one (sprintf, printf, scanf etc.).

[3]from Advances in Format String Exploitation

Part II

Invalid pointer dereferences

NULL pointer dereferences

- Example:

```
x = userinput();
p = malloc(x * sizeof(struct FILE_OPS));
fop = &(p[FILE_TYPE]);
fop->remove(file);
```

- What if the call to `malloc` fails?
- This is a dangerous bug for system call code (in systems that have a unified address space between kernel and userland) as a user process may map the first pages of memory to malicious executable code.
- Proposal: Always check the return value of memory allocation functions.
- Proactive measure for Kernel bugs: prohibit access to the first few memory pages.

Use after free

- A previously allocated object is referenced after deallocation

```
CVE-2013-4560 for lighttpd:

if (!dir_node) {
   if (0 != FAMMonitorDirectory(...)) {
       /* fam_dir de-allocation */
       fam_dir_entry_free(&sc->fam, fam_dir);
   } else {
       sc->dirs = splaytree_insert(sc->dirs, dir_ndx, fam_dir
   }
} else {
   fam_dir = dir_node->data;
}

if (fam_dir) {
    sce->dir_version = fam_dir->version;
}
```

Use after free

- An attacker may be able to control data in the arena of the dynamic allocator
    - **after** the deallocation but **before** the reference
- The attacker may achieve code execution if the reference is a call to a function pointer
- Proposal: Use best practices to produce clean object tracking code
    - Allocate and deallocate objects within the same code / context
    - Use a single policy for deallocation (free() and set to NULL) and existence check (check for NULL)
    - Also, use static + dynamic analysis tools to identify use-after-free bugs

Heap spraying and Heap Feng Shui

- Use after free bugs are very common in web browsers
  - e.g. an HTML component is removed from the DOM (via javascript) but is later referenced (via javascript)
- To gain reliable exploitation, attackers use the following techniques:
  - Heap spraying
    1. Fill a large portion of the browser allocator's arena with shellcode (e.g. 200MB)
    2. Overwrite the function pointer with an address that is likely to fall on the shellcode (e.g. `0x0c0c0c0c`). By the way `0x0c0c0c0c` itself can act as a nopsled (`or al, 0x0c`) !
  - Heap Feng Shui
    1. The browser's allocator is deterministic
    2. Trigger specific allocations and deallocations to produce consecutive blocks (i.e. blocks that don't have metadata or other objects between them)
    3. Make exploitation more reliable by writing the shellcode to these consecutive blocks
- For more info see Alexander Sotirov's "Heap Feng Shui in JavaScript" presentation

Double free

- A special case of a use-after-free bug
- The program erroneously calls a second free() on an already free'd object
- The attacker will try to overwrite the allocator's metadata
  - **after** the first free but **before** the second free
- Due to the mechanism of putting a newly free'd chunk to the free list the attacker can trigger an arbitrary write to process memory
  - For example, an arbitrary write to the `.dtors` section where code will later be executed before the program exits
- Proactive protection: 'Safe unlinking' techniques can be used by deallocators to test chunk pointers before doing a free()
- Proposal: Use best practices for object tracking code

Part III

Integer-related issues

Signedness issues

- Example:

```
int x = userinput();
if (x < MAX_SIZE) {
    memcpy(dest, src, x);
} else {
    /* Handle the error condition ... */
}
```

- memcpy expects an unsigned integer for its 3rd parameter
- x is a signed integer that will be casted to an unsigned one
- e.g. -1 will be casted to 4,294,967,295
- Proposal: Avoid using signed integers where unsigned ones are required (and the reverse). Perform strict integer validation checks.

Integer Overflows

- Example:

```
unsigned int num = userinput();
p = malloc(num * sizeof(struct HEADER));
for(i=0; i<num; i++) {
   p[i] = src[i];
}
```

- What happens if `num * sizeof(struct HEADER)` is larger than the maximum integer value?
- Proposal: In integer arithmetic take into account overflows and underflows.

Integer Overflows

- Checking for an overflow:

```
if ((a+b) < a) {
    ...
}
```

- Checking for an underflow:

```
if ((a-b) > a) {
    ...
}
```

Part IV

Memory and Information Leaks

Memory leak

- A bug in memory management that eventually eats up the available process memory (denial of service)
- Consider the case where an attacker may influence the `hdr.sections` value in the code below:

```
for(i=0; i<hdr.sections; i++){
     cur_section->next = malloc(SECTION_SIZE);
     cur_section = cur_section->next;
}
```

- Proposal: Whenever possible, limit the allocation size and number of allocated chunks that are associated with a user action.

Information leak

- A bug causing the unintended exposure of information
- We'll focus on data coming from application or kernel memory
- Example:

```
write(sock, buf, nbytes);
```

- What happens if an attacker can control the `nbytes` variable, providing a value that is larger than the size of `buf` ?
- Information leaks such as this may expose:
  - sensitive data (e.g. private keys, passwords etc.)
  - security tokens (e.g. canary values)
  - memory addresses (i.e. deduce ASLR mappings)
- Proposal: Always make sure that the data being copied from a buffer are less or equal to the data available in the buffer. Do buffer initialization with `calloc`, `memset` etc.

Pointer obfuscation

- It is very common for an API to return the address of an object as the object identifier
- Wait, that's an information leak!
- Solution: pass an obfuscated pointer to the user
- Once the obfuscated pointer is passed back to the API, it is deobfuscated
- An example from the Linux kernel:

```
static long kptr_obfuscate(long v, int type)
{
    return (v ^ cookies[type][0]) * cookies[type][1];
}
```

Part V

Minimizing the vulnerability handling work

Software Design vs Memory Corruption Vulnerabilities

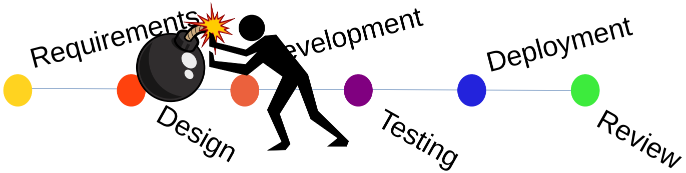
- Memory-related vulnerabilities stem from bad (or difficult) design decisions
- They provide us with a perfect example of vulnerabilities we are exposed to when we choose a certain ecosystem of technologies
- Let’s discuss how we could go about minimizing this vulnerability handling work

Minimizing the vulnerability handling work

- To minimize the risk and costs associated with vulnerability handling, a project needs to follow three parallel strategies:
  - Identify and fix security defects as early as possible
  - Eliminate security defects and related bug classes by design
  - Prefer third party components offering better security

The Shift Left Paradigm

- As security practices come with considerable cost, they are often left to the last minute (if ever performed)
  - e.g. Threat Modeling after the project prototype has been formed
  - e.g. Manual Security Testing after the product has been released
- The *Shift Left* paradigm highlights the advantages of performing security activities early on in the SDLC so as to minimize project risks
  - Security Requirements Analysis
  - Early Threat Modeling or Design Review
  - Source Code Auditing

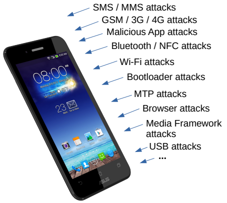


Eliminating security defects by design

- We will cover two approaches for eliminating security defects through design efforts
  - Attack Surface Reduction
  - Using Better Building Blocks

Attack Surface Reduction

- The *Attack Surface* of a target system consists of the points that an attacker may abuse to achieve a goal
- In the Design/Implementation Phase of projects, it is customary to *reduce the software attack surface*
  - Remove unnecessary components / functionalities
    - See wolfi (container) and unikernel (VM) runtimes
  - Employ authentication and authorization checks
  - Execute flows under the *Principle of Least Privilege*
    - Grant flow the least privilege possible to perform its task
  - Stop further compromise through *Zero Trust* Model
    - Containerize components; assume any component may be compromised.
    - Always perform explicit authentication and authorization checks on all available data points; do not consider any peer component / user-flow as trusted.



Using Better Building Blocks

It is possible to eliminate certain types of bugs by carefully choosing the ingredients (“building blocks”) of the project

- Formally Verified Algorithms
- Memory-safe Languages
- LangSec and Framework-based security[4]

[4]we will discuss these topics later in the lectures

Formally Verified Algorithms

- Algorithms have mathematical properties that can be proven to hold true
  - We can exploit this characteristic of algorithms to prove that a certain implementation carries *all the desired properties* of a certain *model*[5]
  - *Automated verification* usually involves *symbolic execution* and SAT/SMT *constraint solving*
- Problems
  - Real world programs have complex constructs that are not easily modelled by mathematical formulas (e.g. side-effects such as printing to the console)
  - Real world programs may require vast amounts of memory to analyze (e.g. state explosion of a loop depending on a variable)
- We generally “relax” the constraints by ignoring/disabling some parts of the program or by simplifying/transforming some parts of the program

[5]See seL4 for a formally verified microkernel, or s2n for formally verified TLS crypto

Memory-safe Languages

- Systems software (bootloaders, kernels, libraries, services, performance-critical applications etc.) have been plagued for decades by memory corruption bugs (e.g. buffer overflows, double frees etc.)
  - Such software typically requires direct memory access for hardware manipulation or to achieve high performance
  - It is highly likely that large pieces of software written in languages with direct memory access capabilities will carry memory corruption bugs
- Memory-safe languages such as Python, Java and Rust default to not allowing the programmer to access memory directly (but only through an explicit escape hatch) and are thus not prone to these types of bugs
- Rust[6] in particular, comes with such a minimal runtime (similar to C) that makes it also ideal for Systems Programming tasks
  - The Rust compiler performs simplified[7] *static analysis* on code and errors out if the code would lead to memory violations or race conditions

[6] `https://www.rust-lang.org/`

[7] Rust code comes with variable ownership and lifetime restrictions

Choosing Third Party Components through Security Metrics

- How do you evaluate third party components based on security metrics? This question poses an open problem!
  - Use software quality metrics[8] (Isn't security $\subset$ quality?)
  - Rate the software's security design decisions?
  - Count instances of known but unfixed vulnerabilities?
  - Run static / dynamic analysis and count potential vulnerabilities?
  - Run exploratory testing (e.g. fuzz testing) and measure the coverage?
  - Measure the mean-time between a security defect report and its patch?
  - Measure the reported number of vulnerabilities?
  - Perform a code audit?
  - Measure the time since the last (complete) audit?
  - Measure the parts of the code that have not been audited?
  - Count sec. best practices employed through automation (CI/CD)?
- There is an initiative[9] that attempts to provide a scorecard for opensource software by the *OpenSSF* project of the Linux Foundation

[8] see Measurement Based Open Software Quality Evaluation, by Samoladas, Spinellis et al.
[9] `https://securityscorecards.dev`

Part VI

Conclusion

Recap: Important Concepts

- Memory Safety and Memory Safe Languages
- Buffer Overflows
- Invalid Pointer Dereferences
- Integer Overflows
- Memory vs Information Leaks
- Shift Left paradigm
- Attack Surface Reduction
- Principle of Least Privilege and Zero Trust paradigms
- Formally Verified Algorithms

Further reading material

- *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*
- *The shellcoder's handbook: discovering and exploiting security holes, 2^{nd} edition*
- *Hacking: The Art of Exploitation (2nd Edition)*
- *Blue Fox: Arm Assembly Internals and Reverse Engineering*

Questions?