revision of **arrays**

8 weeks of revision to learn what an **array** is

Exam

- **Date** Dec 5
- **Time** Morning
  - 15 min reading time + 3 hour exam
  - Will have more details via Emails
- **In Person** (I'm not sure where yet - 3 rooms booked)
- **Done on your own machines!**
  - **You can't do this from home, bring laptop to University**
  - **Digital Ocean droplet?**
  - **SETUP YOUR TOOLING BEFORE THE EXAM.** NO EXCUSES
  - **If this is an issue tell me before end of week.**
- **Similar format to midterm**

More details

https://exam.comp6447.lol/

3 sections. Each worth 33% of the exam. 1% for following instructions

- Exploitation
- Reversing
- Source Auditing

4 questions per section

HURDLE: You must solve at least 1 challenge in each section to pass

Will release past exam as the wk10 wargames.

PWN

4 questions

Ordered in difficulty

Similar in style to the midterm

May test stuff not covered in wargames *hint hint*

Src

- Be succinct
  - Extra meaningless/non vulnerable bugs/warnings == less marks
  - If you spend 500 words explaining a simple bug, do you understand it?
- Only one bug per challenge
  - Only post if you are confident it is the exact bug
- Must give example of how the bug can be used
  - Not necessarily a payload

Src

Valid solution

- **len** is a user controlled signed int
- If len is set to a large number, we can bypass the initial length check with an integer overflow.
  - EXAMPLE: header_size + INT_MAX
  - This would result in an overflown **negative number**
- read() takes in a size_t unsigned integer
  - len+header_size would be treated as a large number instead of negative
- Allows an attacker to perform a buffer overflow on `storage`

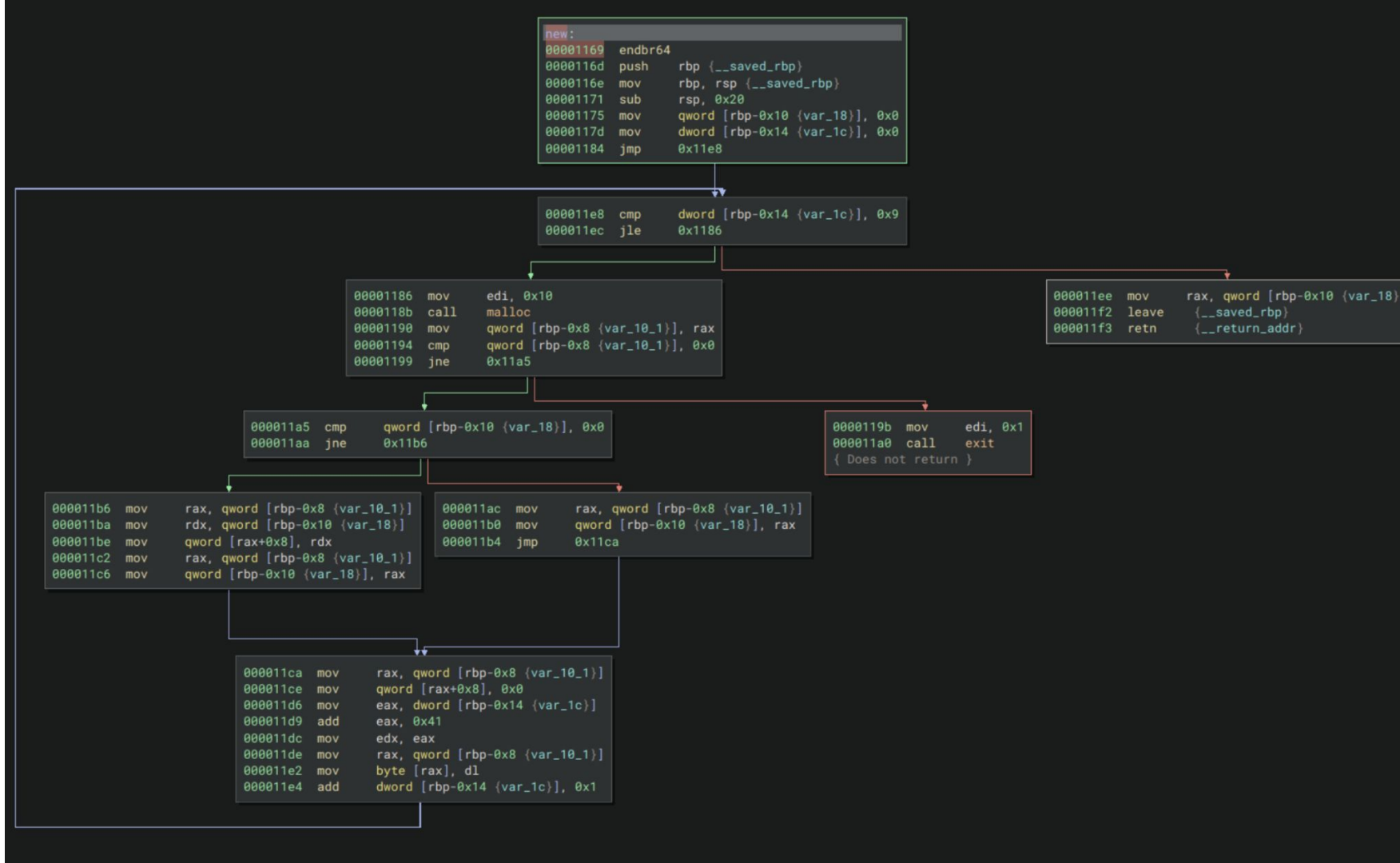
```
if (header_size + len > 32) {
  printf("no");
  return 1;
}

read(0, storage, len + header_size);
```

Re

- Calling stuff `var_20, var_16` == no marks for the challenge
  - Understanding intent is important as code structure. Give variables good names
- Make sure code makes sense, compiles and has good indentation
- One of the questions may ask you to find a bug in the resultant code.

re4


new:
00001169 endbr64
0000116d push rbp {__saved_rbp}
0000116e mov rbp, rsp {__saved_rbp}
00001171 sub rsp, 0x20
00001175 mov qword [rbp-0x10 {var_18}], 0x0
0000117d mov dword [rbp-0x14 {var_1c}], 0x0
00001184 jmp 0x11e8
000011e8 cmp dword [rbp-0x14 {var_1c}], 0x9
000011ec jle 0x1186
00001186 mov edi, 0x10
0000118b call malloc
00001190 mov qword [rbp-0x8 {var_10_1}], rax
00001194 cmp qword [rbp-0x8 {var_10_1}], 0x0
00001199 jne 0x11a5
000011ee mov rax, qword [rbp-0x10 {var_18}]
000011f2 leave {__saved_rbp}
000011f3 retn {__return_addr}
000011a5 cmp qword [rbp-0x10 {var_18}], 0x0
000011aa jne 0x11b6
0000119b mov edi, 0x1
000011a0 call exit
{ Does not return }
000011b6 mov rax, qword [rbp-0x8 {var_10_1}]
000011ba mov rdx, qword [rbp-0x10 {var_18}]
000011be mov qword [rax+0x8], rdx
000011c2 mov rax, qword [rbp-0x8 {var_10_1}]
000011c6 mov qword [rbp-0x10 {var_18}], rax
000011ac mov rax, qword [rbp-0x8 {var_10_1}]
000011b0 mov qword [rbp-0x10 {var_18}], rax
000011b4 jmp 0x11ca
000011ca mov rax, qword [rbp-0x8 {var_10_1}]
000011ce mov qword [rax+0x8], 0x0
000011d6 mov eax, dword [rbp-0x14 {var_1c}]
000011d9 add eax, 0x41
000011dc mov edx, eax
000011de mov rax, qword [rbp-0x8 {var_10_1}]
000011e2 mov byte [rax], dl
000011e4 add dword [rbp-0x14 {var_1c}], 0x1

Re

Valid solution

- Shows struct type
  - Useful names (str, next, etc)
- Shows defaults (ie: NULL instead of 0) for pointers
- Good var names (int i, struct head, next)
- Struct dereferences/etc
- If statements concisely written
- Return value

```
struct somestruct {
  char str;
  struct somestruct *next;
};

struct somestruct *new () {
  struct somestruct *head = NULL;

  for (int i = 0; i < 10; i++) {
    struct somestruct *some = malloc(sizeof(struct somestruct));

    if (some == NULL) {
      exit(1);
    }

    if (head == NULL) {
      head = some;
    } else {
      some->next = head;
      head = some;
    }

    some->next = NULL;
    some->str = 'A' + i;
  }

  return head;
}
```

Wargames + fuzzer

- Marking for most wargames is done
- Weeks 1-5 were pretty good
- Half of the course hasn’t submitted weeks 7+ (ROP, heap)
  - hint hint

- Now we start revision
  - **Go back and do every wargame that you skipped**
  - This is the best way to study for final

This lecture

Going to be revision of every week

- This is a help-session Q&A style revision.

If you have **any** questions about **any**thing i say

Stop me and ask…

If you dont stop me and ask when you are confused don’t blame me **when** you fail the exam

Get ready to watch a lecture irl at 1.5x speed

Week 0 - **Arrays**

- I’ll give you a short and a long answer to this question.
-
- **Short Answer:**
  - An array can be defined as an **ordered** collection of **items** indexed by **contiguous integers**.
- **Long answer:**
- **Arrays** a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Week 1 - **Arrays** in C

Didn’t do much. <insert important rant about getting good with tooling>

<replace this with a talk on which Tooling in exam you can use>

The COMP6447 exam is as hard as you make it. If you don’t know how to use your tooling, it will be harder.

Week 2 - buffer overflows into **arrays**

- Dangerous functions
  - gets fgets strcpy
- Things to know how to do
  - Using cyclic or gdb to find distance to ret addr
  - Reversing a binary to understand the size of buffers / difference between buffer and RIP
- What can we do?
  - Overwrite important variables/structures or return address
  - Point RIP to win functions
  - Overwrite variables on the stack

Week 2 - stack canaries at end of **arrays**

- Some magic number pulled in from libc
- Randomised on program startup
  - Stays the same if you fork (can attack web servers)
- 32 bit have two options
  - If local, brute force is an option
  - Need a leak
- 64 bit.. Can’t bruteforce
  - Need a leak
- Always ends in null byte.. (why?)

Week 2 - Reverse engineering **arrays** in x86_64

- Calling convention
- Function prologue/epilogues
- Some instructions
  - Mov, lea, jumps, cmp
- REP instruction
- How does a loop look like in C vs asm?
- RBP offset vs RSP offset
- Disassemblers vs debuggers
- Recognising patterns
  - moving char vs moving int
  - Signed vs unsigned

Week 3 - Reverse engineering

- Top down vs bottom up approach
- Do we want to look at a high level and look for design flaws
- Do we want to look at individual functions and find exploits
  - Look at where input is/ where it goes
- Strace + ltrace
- How do you recognise data structures
  - Arrays
  - Structs
  - Pointers
  - Ints
  - Chars

Week 3 - Shellcoding **arrays**

- What does shellcode do?
  - Mov rax, x; syscall
- When can we actually use shellcode?
  - Performing “Recon”
- Why would we need a NOPsled?
- Egghunter? Syscall proxy?
- Bypassing filters
  - Learn to do this ***hint***
  - What happens if we can’t have this in our payload
    - Newlines
    - Certain bytes
    - Ascii only payload??

Week 4 - Format Strings to create dynamic **arrays**

- Rarely found in wild because GREP exists
- Can be used to do two main things
  - Leak sensitive data
  - Write to places
  - Read/Write primitive
- What is the $
- Step 1) Find location of buffer on stack…
  - Helps us craft arbritrary pointers
- Step 2) Leak / Write data to there
- What does %x do?
- What does %s do?
- What does %n do?
  - What does hhn do? And why do we care?
- What can we overwrite? (got/function pointers/return address??)

Week 5 - Source code auditing to find **arrays**

- Bad API usage
  - Think memset, strncpy(dst, src, strlen(src))
  - Format strings
  - Gets
  - Leaking stuff with strncpy (it doesnt set a NULL byte)
- Heap
  - UAF/Double Free / Custom malloc implementations?
- Logic bugs
- Integer overflows / Underflows
  - Can lead to buffer overflows in the future,
  - Or incorrect program state
- Type conversions
  - Converting between char and an int
  - Signed to unsigned
  - Pointers to float… etc

Week 5 - Source code auditing

- Using sizeof incorrectly
- Pointer arithmetic, char* vs int* and ++
- Race conditions
  - Not using locks
- **A big one is forgetting early exits/returns on errors**
- Similar to RE, top down vs bottom up approach

Week 5 - Fuzzing **arrays**

- Probably won’t be in the exam
- While we are here..
  - How are assignments going?
    - Don’t forget about them
- Something awesomes for free bonus marks?
- Do something unique

Before we go into the later topics

Any questions?

ROP/HEAP is just a mixture of the previous weeks content

Week 7 - ROP chaining **arrays** together

- Why do we use ROP?
- Ret2code vs ret2libc vs pure ROP
- What can we do by Chaining functions
  - Using mprotect to get working shellcode
- Pop pop ret
- What happens if you can’t find any good gadgets??
  - Leak libc
- If NX and no win functions
  - Will either be ret2libc or ROP
- Will 100% be in final exam
- Stack pivots
  - Very useful
  - Ropper -f file --stack-pivot

Week 7 - ROP

- Libc-database
- How to get a leak if you have ROP
  - puts(puts)
  - puts@plt (puts@GOT)
  - puts(*puts)
- Retsled vs Nopsled
  - Why would we need this?
- More on pivoting..
  - Xchg instruction
  - Add esp, …
  - Ret <xxx> instruction
- More advanced ROP? (SROP, One Gadget)

Week 8 - HEAPs of **arrays**

- What is the role of malloc/free
- The malloc chunk is important to understand
- Lower 3 bits of size representing things
  - Why are chunks multiples of 8

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
 struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
 struct malloc_chunk* bk_nextsize;
};
```

Week 8 - HEAP exploitation in the **array**

- Difference between tcache fastbins smallbins unsorted bins???
  - Size
  - Coallecse?
  - Speed?
  - Security?
- Usually we group together multiple bugs to exploit a program
- Use after free
  - Either read or write after free.. Or both
  - If read what can we do?
  - If write what can we do?
- Double free
  - How is this useful
  - Overlapping chunks
- Forging chunks. Why? What can we do?

Week 8 - HEAP

- Heap spraying?? Sometimes useful.. Probably won't be done in this course
- One_gadget
- What if one_gadget doesn't work
  - HEAP into ROP
- Use a stack pivot to get a ROP chain going
- If you have a small buffer on stack, but can't overflow
  - Add RSP, 0x30
  - Now esp points into your buffer
  - Roproprop
- Smallbin/Largebin use in exploitation (Pre-tcache or V-Large size chunks)

Week 10 - Revision of **arrays**

- Tooling is important
- Speed is the most important thing for this exam
- Challenges from week 2-5 should be solvable in < 20 min by now
  - If they aren’t go do them over and over
    - Until they are
  - Maybe not the source code or RE ones
    - But the exploitation challenges should mostly be super easy right now
- If you can master ROP and HEAP chals, the exam will be **trivial**
- Will release exam from 202X soon

What now ? - after this course

- **What binary stuff didn’t we cover?**
- Actual logic errors
  - Logic errors in compilers/browsers
- Race conditions
- Anything to do with hacking a kernel
- Automated reversing
  - Angr?
  - z3?
- Automating our exploitation
  - Angrop
- **Fuzzing**
  - **Modern day exploit research relies on fuzzing...**