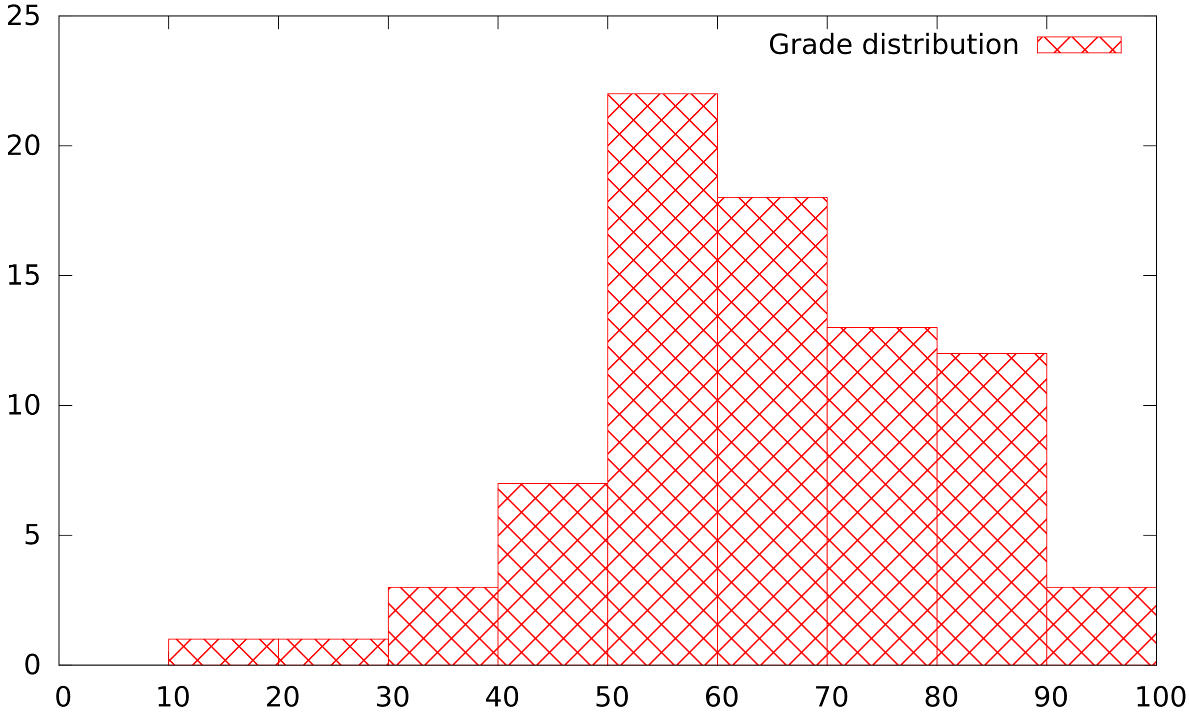
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2014

Quiz I Solutions



Histogram of grade distribution

Mean 64.4, Stddev 15.5

I Baggy Bounds and Buffer Overflows

1. [6 points]: At initialization time, a baggy bounds system on a 32-bit machine is supposed to set all of the bounds table entries to 31. Suppose that, in a buggy implementation with a `slot_size` of 32 bytes, bounds table initialization is improperly performed, such that random entries are incorrectly set to 1.

Suppose that a networked server uses an uninstrumented library to process network messages. Assume that this library has no buffer overflow vulnerabilities (e.g., it never uses unsafe functions like `gets()`). However, the server *does* suffer from the bounds table initialization problem described above, and the attacker can send messages to the server which cause the library to dynamically allocate and write an attacker-controlled amount of memory using uninstrumented code that looks like this:

```
// N is the buffer size that the
// attacker gets to pick.
char *p = malloc(N);
for (int i = 0; i < N/4; i++, p += 4) {
  *p = '\a';
  *(p+1) = '\b';
  *(p+2) = '\c';
  *(p+3) = '\d';
}
```

Assume that the server uses a buddy memory allocator with a maximum allocation size of 2^{16} (i.e., larger allocations fail). What is the smallest *N* that the attacker can pick that will definitely crash the server? Why will that *N* cause a crash?

Answer: The uninstrumented library code does not check bounds table entries when it does pointer arithmetic. Thus, the code snippet above is unaffected by the incorrect initialization of the bounds table. However, the code snippet does not check the return value of `malloc()` for NULL; so, the attacker can select an *N* of $2^{16} + 1$, cause `malloc()` to return NULL, and get the server to crash.

2. [6 points]: Modern CPUs often support NX ("no execute") bits for memory pages. If a page has its NX bit set to 1, then the CPU will not run code that resides in that page.

NX bits are currently enforced by the OS and the paging hardware. However, imagine that programs execute on a machine whose OS and paging hardware do not natively support NX. Further imagine that a compiler wishes to implement NX at the software level. The compiler associates a software-manipulated NX bit with each memory page, placing it at the bottom (i.e., the lowest address) of each 4KB page.

The compiler requires that all application-level data structures be at most 4095 bytes large. The compiler allocates each stack frame in a separate page, and requires that a stack frame is never bigger than a page. A stack frame might look like the following:

```
                        ...
             |-----------------------|
             |                       |
             +-----------------------+
entry %esp-->|      return address   |
             +-----------------------+
new %ebp---->|      saved %ebp       |
             +-----------------------+
             |      buf[4]           |
             |      buf[3]           |
             |      buf[1]           |
             |      buf[0]           |
             +-----------------------+
             | ...other stack vars...|
             +-----------------------+
new %esp---->|       NX bit          |
             +-----------------------+
```

such that, as shown in the sample code above, an overflow attack in the frame will not overwrite the frame's NX bit.

The compiler also associates NX bits with each normal code page. The NX bit for a stack frame is set to "non-executable", and the NX bit for a normal code page is set to "executable".

The compiler instruments updates to the program counter such that, whenever the PC migrates to a new page, the program checks the NX bit for the page. If the bit indicates that the page is non-executable, the program throws an exception.

Describe how a buffer overflow attack can still overwrite NX bits.

Answer:

- A buffer overflow in the currently active frame can spill into the frame that is above it in RAM. Thus, a callee can overwrite its caller's NX bit.
- If a buffer overflow can corrupt a pointer value, the attacker can make the pointer point to the address of an NX bit. If that pointer is dereferenced and assigned to, the NX bit will be overwritten.
- The attacker could mount a return-to-libc attack to use preexisting code to reset an NX bit.

II Stack Canaries and Return-Oriented Programming

3. [4 points]: Stack canaries live in an area of memory that programs can read as well as write. In the typical buffer overflow attack (e.g., via the `gets()` function), what prevents an attacker from simply reading the canary value and then placing that canary value in the overflow payload?

Answer: In the typical buffer overflow attack, the attacker cannot execute arbitrary code; instead, the attacker can only supply inputs that the program will not bounds-check during a copy operation. Thus, the attacker can only *write* the stack. In other words, vulnerable functions like gets() do not allow the attacker to directly read a value and then insert that value into the attack payload.

You get partial credit if you say that the canary might contain terminating characters, such as '`\0`', that stop `gets()` from reading beyond the point.

4. [10 points]: In the first part of a BROP attack, the attacker must find gadgets that pop entries from the stack and store them into attacker-selected registers. Suppose that the attacker has already found the address of a stop gadget and a trap value (i.e., a memory value which, if accessed, causes a fault). In the stack diagram below, depict what a buffer overflow should write on the stack to identify pop gadgets which pop exactly two things from the stack e.g., `pop rdi; pop rsi; ret;`. If it doesn't matter what goes in a particular memory location, put "Doesn't mattter". To represent the values for the stop gadget and the trap, simply write "stop" or "trap". To represent the address of a candidate pop gadget, write "probe".

```
                        ...
             |-----------------------|
             |                       |  Value: ________
             |-----------------------|
             |                       |  Value: ________
             |-----------------------|
             |                       |  Value: ________
             |-----------------------|
             |                       |  Value: ________
             |-----------------------|
             |                       |  Value: ________
             |-----------------------|
             |                       |  Value: ________
             +-----------------------+
entry %esp-->|      return address   |  Value: ________
             +-----------------------+
new %ebp---->|      saved %ebp       |  Value: ________
             +-----------------------+
             |      buf[3]           |  Value: ________
             |      buf[2]           |  Value: ________
             |      buf[1]           |  Value: ________
             |      buf[0]           |  Value: ________
             +-----------------------+
new %esp---->| ...other stack vars...|
             +-----------------------+
```

```
                           ...
             |-----------------------|
             |                       |  Value: trap
             |-----------------------|
             |                       |  Value: trap
             |-----------------------|
             |                       |  Value: trap
             |-----------------------|
             |                       |  Value: stop
             |-----------------------|
             |                       |  Value: trap
             |-----------------------|
             |                       |  Value: trap
             +-----------------------+
entry %esp-->|      return address   |  Value: probe
             +-----------------------+
new %ebp---->|      saved %ebp       |  Value: Doesn't matter
             +-----------------------+
             |      buf[3]           |  Value: Doesn't matter
             |      buf[2]           |  Value: Doesn't matter
             |      buf[1]           |  Value: Doesn't matter
             |      buf[0]           |  Value: Doesn't matter
             +-----------------------+
new %esp---->| ...other stack vars...|
             +-----------------------+
```

Answer:

III OKWS and OS Security

Suppose Unix did not provide a way of passing file descriptors between processes, but still allowed inheriting file descriptors from a parent on `fork` and `exec`.

5. [4 points]:

What aspects of the OKWS design would break without file descriptor passing?

(Circle True or False for each choice.)

A. True / False It would be impossible for services to send messages to `oklogd`.

Answer: False.

B. True / False It would be impossible for services to get a TCP connection to a database proxy.

Answer: False.

C. True / False It would be impossible for services to get a TCP connection to the client web browser.

Answer: True.

D. True / False It would be impossible for `okd` to run as a non-root user.

Answer: False.

Consider the following Python code for a program that might run every night as root on a Unix machine to clean up old files in `/tmp`. The Python function `os.walk` returns a list of subdirectories and filenames in those subdirectories. It ignores `"."` and `".."` names. As a reminder, a Unix filename cannot contain `/` or NULL bytes, and `os.unlink` on a symbolic link removes the symbolic link, not the target of the symbolic link.

```
def cleanup():
  ## Construct a list of files under /tmp that are over 2 days old.
  files = []
  for (dirname, _, filenames) in os.walk('/tmp'):
    for filename in filenames:
      fn = dirname + '/' + filename
      if os.path.getmtime(fn) < time.time() - 2 * 86400:
        files.append(fn)

  for fn in files:
    os.unlink(fn)
```

6. [10 points]:

Explain how an adversary could take advantage of this program to delete `/etc/passwd`.

Answer: The adversary can exploit a race condition. First, create a directory `/tmp/foo` and a file `/tmp/foo/passwd` in there, and set the modification time of `/tmp/foo/passwd` to be over 2 days old. Then wait for the script to run `os.walk` and now delete `/tmp/foo/passwd` and `/tmp/foo`, and create a symlink `/tmp/foo` to `/etc`. Now the `cleanup` code will run `os.unlink("/tmp/foo/passwd")`, which will remove `/etc/passwd`.

We also gave partial credit to the answer of creating a symlink `/tmp/foo` pointing at `/etc`, under the assumption that `os.walk` follows symlinks, even though in reality it does not.

IV Native Client

Answer the following questions about how Native Client works on 32-bit x86 systems, according to the paper "Native Client: A Sandbox for Portable, Untrusted x86 Native Code."

7. [6 points]: Which of the following statements are true?

(Circle True or False for each choice.)

A. True / False The Native Client compiler is trusted to generate code that follows Native Client's constraints.

Answer: False.

B. True / False The Native Client validator ensures that no instruction spans across a 32-byte boundary.

Answer: True.

C. True / False The Native Client service runtime is checked using the validator to ensure its code follows the constraints.

Answer: False.

D. True / False Native Client requires additional instructions before every direct jump.

Answer: False.

E. True / False Native Client requires additional instructions before every indirect jump.

Answer: True.

F. True / False Native Client requires additional instructions before every memory access.

Answer: False.

8. [6 points]:

For the following x86 code, indicate whether Native Client's validator would allow it (by writing **ALLOW**), assuming the parts after `...` are valid, or circle the *first* offending instruction that causes the validator to reject the code.

```
10000:      83 e0 2e             and    $0x2e,%eax
10003:      40                   inc    %eax
10004:      01 ca                add    %ecx,%edx
10006:      4a                   dec    %edx
10007:      eb fa                jmp    0x10003
10009:      b9 ef be ad de       mov    $0xdeadbeef,%ecx
1000e:      8b 39                mov    (%ecx),%edi
10010:      8b 35 ef be ad de    mov    0xdeadbeef,%esi
```

```
   10016:       8b 66 64                    mov    0x64(%esi),%esp
   10019:       5b                          pop    %ebx
   1001a:       8b 58 05                    mov    0x5(%eax),%ebx
   1001d:       83 e0 e0                    and    $0xffffffe0,%eax
   10020:       ff e0                       jmp    *%eax
   10022:       f4                          hlt
...
```

Answer: The validator will complain about the jmp at 0x10020 with error “Bad indirect control transfer”; see Figure 3 in the NaCl paper.

V Symbolic execution

Consider the following Python program running under the concolic execution system from lab 3, where `x` is a concolic integer that gets the value 0 on the first iteration through the loop:

```
def foo(x):
  y = x + 7
  if y > 10:
    return 0
  if y * y == 256:
    return 1
  if y == 7:
    return 2
  return 3
```

9. [6 points]:

After running `foo` with an initial value of `x=0`, what constraint would the concolic execution system send to Z3 for the second `if` statement?

Answer: (x+7)*(x+7)=256 AND NOT (x+7)>10

More precisely, in Z3's s-expression:

(and (= (> (+ x 7) 10) false) (not (= (= (* (+ x 7) (+ x 7)) 256) false))).

VI Web security

10. [8 points]: Suppose that a user visits a mashup web page that simultaneously displays a user's favorite email site, ecommerce site, and banking site. Assume that:

- The email, ecommerce, and banking sites allow themselves to be placed in iframes (e.g., they don't prevent this using `X`-Frame-Options headers).
- Each of those three sites is loaded in a separate iframe that is created by the parent mashup frame.
- Each site (email, ecommerce, banking, and mashup parent) come from a different origin with respect to the same origin policy. Thus, frames cannot directly tamper with each other's state.

Describe an attack that the mashup frame can launch to steal sensitive user inputs from the email, ecommerce, or banking site.

Answer: The parent mashup frame can place a invisible iframe atop (say) the banking site. Using this invisible frame, the mashup can steal the user's keypresses as she tries to enter her login name and password.

Additional attacks are possible. For example, if the user allows the mashup frame to do screensharing, the mashup frame can take a snapshot of child frame content and send that snapshot to an attacker-controlled server; this allows the attacker to (for example) see emails that the user is currently composing. The mashup frame can also exploit a child frame that does improper `postMessage()` validation and responds to requests from arbitrary initiators.

11. [8 points]: Each external object in a web page has a type. That type is mentioned in the object's HTML tag (e.g., an image should have an `<img>` tag like `<img src="http://x.com/x.gif">`). An object's type is also described as a MIME type in its HTTP response (e.g., `Content-type: "image/gif"`).

These two kinds of type specifications can mismatch due to programmer error, misconfiguration, or malice. For example, for the tag `<img src="http://x.com/x.gif">`, the server might return the MIME type "text/css".

Suppose that, in the case of a type mismatch, the browser uses the MIME type in the HTTP response to determine how to interpret an object. For example, if X's frame tries to load the MIME-type-less tag `<img src="http://Y/file">`, and Y's server returns a MIME type of "text/css", the browser will interpret the fetched object as CSS in X's frame, even though the object is embedded in X's frame as an `<img>` tag.

Why is this a bad security policy?

Answer: The security policy is bad because origin X can include what it believes to be passive content (e.g., an image), but origin Y can convince the browser to interpret that content as Javascript code! That JavaScript code will be supplied by Y, but it will run with the authority of X.

12. [6 points]: In a SQL injection attack, attacker-controlled input is evaluated in the context of a SQL query, resulting in malicious SQL statements executing over sensitive data. Ur/Web allows web applications to directly embed SQL queries in a page; furthermore, those queries may contain information that originates from the user or an untrusted source. Why is this safe in Ur/Web?

Answer: Ur/Web is a strongly-typed system which does not allow external strings to be directly (and maybe accidentally!) interpreted as executable code, SQL queries, etc. This contrasts with the standard web world, in which it is not obvious whether it is safe to (say) assign an externally-supplied string to the innerHTML property of a DOM node.

VII Network security and Kerberos

Ben Bitdiddle is concerned about the sequence number guessing attack that Steve Bellovin described in section 2 of his paper, where an adversary can spoof a TCP connection to a server from an arbitrary source IP address, and send data on that connection.

Ben implements the following strategy that his server will use for choosing the initial sequence number ISN_s of an incoming TCP connection:

$$\text{ISN}_s = \text{ISN}_{\text{original}} \oplus \text{IP}_{\text{src}} \oplus \text{IP}_{\text{dst}} \oplus (\text{Port}_{\text{src}} || \text{Port}_{\text{dst}}) \tag{1}$$

where $\oplus$ refers to the XOR operation and $||$ refers to concatenation; the IP fields being XORed refer to the 32-bit IP addresses of the source and destination of the TCP connection; and the Port fields refer to the 16-bit source and destination ports. Assume $\text{ISN}_{\text{original}}$ increments by 64 for each new incoming connection, and initially starts at some random value.

13. [8 points]:

Explain how an adversary could still launch a sequence-number-guessing attack against Ben's server with a small number of tries.

Answer: Send two connection requests (SYN packets) to Ben's server, back-to-back: one from the adversary's own IP address, and one from the spoofed source IP address. Let's send the connection request from the adversary's own request first. Then, when the SYN-ACK arrives to the adversary, recover the corresponding $\text{ISN}_{\text{original}}$ by XORing with the source and destination IP and port numbers. Then reconstruct the $\text{ISN}_{\text{original}}$ that the spoofed connection would get (+64), and XOR with the spoofed source and destination IP and port. Use that result when sending the ACK for the spoofed connection.

Suppose the KDC server at MIT developed a subtle hardware problem, where the random number generator became highly predictable (e.g., it would often produce the same result when asked for a "random" number).

14. [6 points]:

How could an adversary leverage this weakness to access some user's data on a file server that uses Kerberos for authentication? Describe the minimal amount of additional access the adversary might need to mount such an attack. Assume the file server ignores IP addresses in Kerberos tickets, and that the keys of all principals were generated *before* the server developed this hardware problem.

Answer: Observe at least one message from victim to file server, and extract the user's ticket from that packet. Use the knowledge of the RNG predictability to guess the corresponding $K_{c,s}$. Now use the ticket and the guessed $K_{c,s}$ to issue arbitrary requests to the file server.

VIII 6.858

We'd like to hear your opinions about 6.858. Any answer, except no answer, will receive full credit.

15. [2 points]: We introduced a new lab on symbolic execution this semester (lab 3). How would you suggest improving this lab in future semesters?

Answer: 18x iffy instructions, missing/vague specs, more comments in the code; 15x make it more exploratory + open-ended, less pre-defined, have students implement more code; 8x it was possible to do the lab without understanding things; 7x better test cases (e.g., exercise 3); 6x improve instructions for exercise 3 (copy Jon's piazza post); 3x recitation about the lab; 2x find more interesting bugs; 2x more background / docs on Z3; 2x faster test cycle; give fewer hints; implement parts of the SMT solver; examples of how concolic execution would work on some piece of code; better explanation of concolic vs symbolic; more exercises like signed avg; shorter explanations needed for lab; use real web app instead of zoobar; better debugging support; maybe ask students to implement parts of the AST structure?; actually create constraints for Z3; expand exercises 6 and 7 (more invariant checks); better visualizations (borrow Austin's grapher from Commuter); unclear what's in concolic variables; diagrams in lab writeup; add `concolic_int.__rsub__`.

16. [2 points]: Are there other things you'd like to see improved in the second half of the semester?

Answer: 11x more attacks; 5x give hints about hard-to-understand points / background from papers / where to focus, before reading; 4x office hours on weekends / friday; 3x less tedious papers; 3x more feedback on labs; 3x slower-paced lectures/class; 3x less discussion of papers, more discussion of new material; 2x allow submitting paper questions after 10pm; 2x more quiz review sessions; 2x more diagrams / examples; recitations for stuff not covered in lecture; CTFs; more extensive lab test cases; more web security; newer versions of papers/ideas/systems; post lecture notes before lecture; address more questions from paper questions; more OS-level security; do lecture before paper; more late days; stay longer on each given topic; break in the middle of lecture; more analyzing other students' lab code, peer review; more interactive discussions; hands-on exercises for ideas from class; program verification; don't sweep details under the rug; more conceptual readings; in-class demos; more help from TAs on Piazza; fewer labs to give more project time; talk more about papers in lecture; upload lecture videos quicker; dislike ASCII diagrams; coffee in lecture.

17. [2 points]: Is there one paper out of the ones we have covered so far in 6.858 that you think we should definitely remove next year? If not, feel free to say that.

Answer: 16x tangled web (long, many didn't read the whole thing!); 8x ur/web; 7.5x capsicum; 5x django (more context); 4.5x nacl; 3x BROP; 3x forcehttps; 2x kerberos (update it!); 2x klee; confused deputy; TCP.

End of Quiz

MIT OpenCourseWare
http://ocw.mit.edu

6.858 Computer Systems Security
Fall 2014

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.