Final Exam (Mock)

This is a closed-book test. You may not refer to any books or notes during this examination, and you may not use any electronic aid.

Write your answers legibly. The intended answers fit within the spaces provided on the question sheets. You may use the back of the preceding page for scratch work. If you run out of room for an answer, continue on the back of the page and clearly mark your answer.

Keep calm! Even though, it feels like you do not know the answer, write your thoughts.

Note: These mock questions are provided to give you an idea about the exam setting, while there is no guarantee for the similarity of the hardness in the real exam. The final will have total 5 questions and duration 60 mins.

Time limit: **50 minutes**.

Write and sign the honor code pledge:

> "*I have neither given nor received unauthorized aid on this examination, nor have I concealed any violations of the Honor Code.*"

_____

_____

_____

_____

(Signature)

_____

(Print your name)

_____

(UA NetID)

1. Short Answer

(a) [2 points] Name two defenses to buffer overflow attacks, and briefly describe how each one works.

(b) [2 points] What anonymity protections does Tor attempt to provide? Be precise.

(c) [2 points] What is HTTP Strict Transport Security (HSTS)? What type of attack does HSTS prevent?

(d) [2 points] What is the same-origin policy? Why is it necessary?

(e) [2 points] Cross-site scripting (XSS) can be used with HTTP _____.
a) DELETE requests
b) POST requests
c) PUT requests
d) All of the above

2. Application Security

You come across the following program:

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

int vuln(char* argv) {
  char buf[4];
  strncpy(buf, argv, 9);
  return sizeof(buf);
}

int main(int argc, char* argv[]) {
  if (argc < 2) {
    printf("Not enough arguments\n");
    return -1;
  }
  int size = -1;
  size = vuln(argv[1]);
  if (size == 24) {
    printf("You Win!\n");
  }
  return 0;
}
```

```
Dump of assembler code for function main:
   0x0804843b <+0>:     push   %ebp
   0x0804843c <+1>:     mov    %esp,%ebp
   0x0804843e <+3>:     and    $0xfffffff0,%esp
   0x08048441 <+6>:     sub    $0x20,%esp
   0x08048444 <+9>:     cmpl   $0x1,0x8(%ebp)
   0x08048448 <+13>:    jg     0x804845d <main+34>
   0x0804844a <+15>:    movl   $0x8048570,(%esp)
   0x08048451 <+22>:    call   0x8048320 <puts@plt>
   0x08048456 <+27>:    mov    $0xffffffff,%eax
   0x0804845b <+32>:    jmp    0x8048491 <main+86>
   0x0804845d <+34>:    movl   $0xffffffff,0x1c(%esp)
   0x08048465 <+42>:    mov    0xc(%ebp),%eax
   0x08048468 <+45>:    add    $0x4,%eax
   0x0804846b <+48>:    mov    (%eax),%eax
   0x0804846d <+50>:    mov    %eax,(%esp)
   0x08048470 <+53>:    call   0x8048414 <vuln>
   0x08048475 <+58>:    mov    %eax,0x1c(%esp)
   0x08048479 <+62>:    cmpl   $0x18,0x1c(%esp)
```

```
   0x0804847e <+67>:    jne    0x804848c <main+81>
   0x08048480 <+69>:    movl   $0x8048585,(%esp)
   0x08048487 <+76>:    call   0x8048320 <puts@plt>
   0x0804848c <+81>:    mov    $0x0,%eax
   0x08048491 <+86>:    leave
   0x08048492 <+87>:    ret

Dump of assembler code for function vuln:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     sub    $0x10,%esp
   0x0804841a <+6>:     mov    0x8(%ebp),%eax
   0x0804841d <+9>:     movl   $0x09,0x8(%esp)
   0x08048425 <+17>:    mov    %eax,0x4(%esp)
   0x08048429 <+21>:    lea    -0x4(%ebp),%eax
   0x0804842c <+24>:    mov    %eax,(%esp)
   0x0804842f <+27>:    call   0x8048350 <strncpy@plt>
   0x08048434 <+32>:    mov    $0x4,%eax
   0x08048439 <+37>:    leave
   0x0804843a <+38>:    ret
```

```
x86 AT&T syntax cheat-sheet:
push   src        // Increment stack appropriate number of bytes and store src there
mov(l) src, dest  // Copy src into dest
and    src, dest  // dest = dest & src
sub    src, dest  // dest = dest - src
cmpl   src1, src2 // Subtracts src2 from src1 and modifies flags: AF CF OF PF SF ZF
jg     target     // jump to target if greater
call   target     // Unconditional branch to target
add    src, dest  // dest = dest + src
jne    target     // jump to target if not equal
lea    src, dest  // dest = keep_as_address(src)
```

(a) [2 points] What is the address of the `ret` instruction that the buffer overflow effects?

(b) [8 points] **vuln** has just been called. The return address is on the stack, and the instruction pointer is at `0x08048414`. At this moment, the **`ebp`** and **`esp`** registers hold the values `0xbffff358` and `0xbffff338`, respectively. Show what happens to the stack below when **vuln** is executed. Stop when the instruction pointer reads `0x0804842f` (don't execute the instruction that calls **strncpy**). Leave any unused cells blank.

Reminder: This is x86 assembly and so bytes are stored in little-endian order.

0xbffff340	↑ main vars ↑				
0xbffff33c	0x60	0xf5	0xff	0xbf	← argv[1]
0xbffff338	0x75	0x84	0x04	0x08	← return address
0xbffff334	0x	0x	0x	0x	
0xbffff330	0x	0x	0x	0x	
0xbffff32c	0x	0x	0x	0x	
0xbffff328	0x	0x	0x	0x	
0xbffff324	0x	0x	0x	0x	
0xbffff320	0x	0x	0x	0x	

3. Web Security

Bob is in his neighborhood coffee shop updating his personal blog, which is hosted on a web server he manages. To navigate to his blog, Bob types http://www.bblog.com into the web browser. The web server responds with a `302` temporary redirect to https://www.bblog.com. The server uses a certificate signed by a trusted CA, but Bob rarely notices the browser lock icon.

(a) [2 points] Name a network-based attack that Mallory could use to gain the ability to view Bob's plaintext traffic, and explain how it works.

(b) [2 points] Name a server-side feature that Bob could deploy to prevent this attack, and explain how it would protect him.

(c) [6 points] Bob is afraid that someone is watching him, so he's built an emergency self-destruct mechanism for his blog. When Bob navigates to a defined endpoint on the server and enters his credentials, all files on the server will be erased. Mallory wants to maliciously trigger this mechanism to destroy the server.

Mallory knows Bob's username is `bobrulez`, but she doesn't know his password. Additionally, Bob has implemented a public comments feature on every blog post, and Mallory knows that the implementation fails to sanitize the text of posted comments.

Below is the web page that Bob created to trigger the self-destruct mechanism:

```
<p>Welcome to Bob's blog. This is the self-destruct button
   for when bad guys are coming for me!</p>
<form action="/kaboom" method="post">
  <input type="text" name="username"></input>
  <input type="password" name="password"></input>
  <input type="hidden" name="csrf" value={{ csrf_token
     }}></input>
  <input type="submit" value="Login"></input>
</form>
```

On the server, the `/kaboom` endpoint is implemented as follows:

```
if request.parameters['csrf'] == request.cookies['
   csrf_token'] and
        is_logged_in_as_admin(request.cookies['session_token
           ']):
  user = sanitize(request.parameters['username'])
  # hexdigest will return a string of characters [0-9a-f]
  password = sha256(request.parameters['password']).
     hexdigest()
  query = "SELECT * FROM users WHERE username='" + user + "
     ' AND password = '" + password + "'"
  SQL.execute(query)
  results = SQL.getResults()
  if len(results) > 0:
    kaboom()
    return '<p>kaboom!</p>'
return '<p>Self-destruct request denied.</p>'
```

where `is_logged_in_as_admin` is a securely implemented function that checks that the browser session has been authenticated as an administrator (e.g., Bob himself). Assume that `csrf_token` is securely generated and difficult to guess.

Bob's sanitize function is implemented as follows:

```
def sanitize(username): # no non-alphanumeric characters
                        # (only numbers and letters allowed
                           )
  i=0
  while i < len(username):
    # if the character is not alphanumeric
    if not username[i].isalnum():
      username = username.remove(i) #remove character at
         index i
    i=i+1
  return username
```

Explain how Mallory can trigger the self-destruct mechanism. Your explanation should include a description of all necessary steps as well as the specific SQL injection string used.

4. More on Web Security

In designing the Death Star, the Galactic Empire decided to use a MySQL database with a PHP backend to store user information. Unfortunately for them, their design had numerous security vulnerabilities. They have hired you as security consultants to ensure their design is correct. Luckily, they didn't hire you to fix their exhaust system.

They had the foresight to use a function, `addSith`, which replaces the characters `'`, `"`, `\`, and `\0` with the string "`sith`".

The PHP backend logic for the login page is as follows:

```
$user = $_POST['user'];
$pass = addSith($_POST['pass']);
$sql = "SELECT * FROM users WHERE name='$user' AND pass='$pass'";
$rs = mysql_query($sql);
if (mysql_num_rows($rs) > 0) {
    // Success
}
```

(a) [3 points] Provide a username/password combination that will always result in a successful login for the user account `sithLord`.

(b) [2 points] Explain how to modify the code to prevent this vulnerability.

(c) [2 points] What is another way, besides input sanitization, to prevent SQL injection?

Darth Vader has decided to run a website on the Death Star's servers that greet users to the Death Star upon entering, via a URL parameter named `user`. The backend code for this page is as follows:

```
$user = $_GET['user'];
echo "<script>
        var name = '$user';
        alert('Welcome, ' + name);
        </script>";
```

(d) [3 points] The URL is of the form https://deathstar.com/greetings?user=DarthSidious. Give a URL that produces the alert "Rule of Two" before anything else is shown to the user.