Paxson
Spring 2017

CS 161
Computer Security

Midterm 1

PRINT your name: ________________________, ________________________
(last) (first)

I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that any academic misconduct will be reported to the Center for Student Conduct, and may result in partial or complete loss of credit.

SIGN your name: ____

PRINT your class account login: `cs161-`______ and SID: ______________________

Your TA's name: ____

Your section time: ____

Exam # for person sitting to your left: ______________

Exam # for person sitting to your right: ______________

You may consult one sheet of paper (double-sided) of notes. You may not consult other notes, textbooks, etc. Calculators, computers, and other electronic devices are not permitted.

You have 80 minutes. There are 5 questions, of varying credit (300 points total). The questions are of varying difficulty, so avoid spending too long on any one question. Parts of the exam will be graded automatically by scanning the **bubbles you fill in**, so please do your best to fill them in somewhat completely. Don't worry—if something goes wrong with the scanning, you'll have a chance to correct it during the regrade period.

If you have a question, raise your hand, and when an instructor motions to you, come to them to ask the question.

Do not turn this page until your instructor tells you to do so.

Question:	1	2	3	4	5	Total
Points:	64	62	58	56	60	300
Score:						

Problem 1 ***True/False*** **(64 points)**

For each of the following, FILL IN THE BUBBLE next to **True** if the statement is correct, or next to **False** if it is not. Each correct answer is worth 4 points. Incorrect answers are worth 0 points. Answers left blank are worth 1 point.

(a) Framebusting allows Javascript in an outer page to access the cookies associated with an inner page loaded in an `iframe`.

○ **True** [✓ ● **False**]

(b) `http://www.coolvids.com:3000/index.html` is in the same origin as `http://coolvids.com:3000/index.html`.

○ **True** [✓ ● **False**]

(c) Browsers apply the Same Origin Policy to determine what URLs can be loaded in `iframe`s.

○ **True** [✓ ● **False**]

(d) If Tyrion uses a browser with no code vulnerabilities and uses a unique, long password for every website he visits, then he will be safe against phishing attacks.

○ **True** [✓ ● **False**]

(e) A recommended defense against clickjacking attacks is for servers to include an HTTP `X-Frame-Options` header in its replies.

[✓ ● **True**] ○ **False**

(f) HTTP-Only Cookies are designed to prevent CSRF attacks.

○ **True** [✓ ● **False**]

(g) An attacker can steal Alice's cookies for `www.squigler.com` by exploiting a buffer overflow vulnerability in Alice's browser.

[✓ ● **True**] ○ **False**

Solution: The key concept underlying this question is that if an attacker can exploit a buffer overflow, they can inject code into Alice's browser that will make the browser do whatever they wish—including transmit cookies to a remote server.

(h) Executable Space Protection (e.g., DEP and W⊕X) is a defense against buffer overflow attacks.

● **True** ○ **False**

(i) ASLR is a defense against buffer overflow attacks that requires operating system support.

● **True** ○ **False**

(j) ASLR will prevent any attack that overflows local variables from executing injected code.

○ **True** ● **False**

(k) Stack canaries are a defense against buffer overflow attacks that requires operating system support.

○ **True** ● **False**

Solution: A compiler can add the canary generation and checking without needing any extra functionality from the operating system.

(l) Stack canaries will prevent any attack that overflows local variables from executing injected code.

○ **True** ● **False**

(m) Stack canaries provide some protection against `printf` format string vulnerabilities, but do not protect against all such vulnerabilities.

● **True** ● **False**

Solution: The original solution stated the following, in support of an answer of **True**:

> `printf` format string vulnerabilities allow attackers to write to the stack. If part of what an attacker writes is a new RIP value to cause a jump to code they've written elsewhere on the stack, then stack canaries will prevent the attack from succeeding.

However, this perspective reflects a variant of stack canaries not discussed in class. (The variant encodes the correct RIP, and in some versions the correct

SFP, into the canary.) As discussed on Piazza, for "vanilla" stack canaries, an attacker who is at all careful can always avoid altering the canary. Thus, the correct answer is **False**, as such canaries do not provide any protection.

This other part of the original solution remains correct regardless of the type of canary:

> On the other hand, other forms of format string vulnerabilities read data from the stack but do not alter stack contents; or write onto the stack in pinpoint locations (for example, to alter the value of a variable). Stack canaries cannot prevent these attacks.

(n) AMD's NX feature, and Intel's similar XD feature, provide protections against XSS attacks.

○ **True** [✓ ● **False**]

(o) If a web page from `abc.com` includes a script from `xyz.com`, the Same Origin Policy puts the script from `xyz.com` in the `abc.com` origin.

[✓ ● **True**] ○ **False**

Solution: When discussing the SOP, we pointed out this important "exception" when loading Javascript. It enables the use of Javascript "libraries". One example we used was how many web sites will include a Google Analytics script, which they load from one of Google's servers (and thus it comes from one of Google's origins).

(p) The Same Origin Policy prevents XSS attacks if a browser implements it correctly.

○ **True** [✓ ● **False**]

Problem 2 ***Multiple Choice*** **(62 points)**

(a) (8 points) Many people lock valuables in a safe in their house in addition to locking the doors of the house. **Mark ONE** security principle that **best** fits with this approach:

○ Ensure Complete Mediation ○ Don't rely on security through obscurity

[✓ ● Defense in Depth] ○ Privilege Separation

Solution: The situation described requires an adversary to work their way past both of two separate defenses.

(b) (8 points) Bob places a duplicate key to his house under one particular stone in his front yard in case he forgets or loses his main key. **Mark ONE** security principle that **best** fits with his approach:

○ Ensure Complete Mediation [✓ ● Don't rely on security through obscurity]

○ Defense in Depth ○ Privilege Separation

Solution: Bob's approach fails if an adversary knows its details.

(c) (10 points) Assume that an airport wants to achieve two security goals: **(a)** passengers can only board planes if they have boarding passes issued in their actual name, and **(b)** passengers cannot board planes unless they have undergone a security inspection by the TSA.

Consider the following design that an airport uses to try to meet these goals. The airport operators arrange that passengers can only board an airplane if they:

1. show a boarding pass and photo ID at a TSA security checkpoint, for which the photo on the ID matches the passenger presenting it, and the name on the ID matches that on the boarding pass
2. pass through a TSA security inspection at that checkpoint
3. present a boarding pass at the gate when actually going onto the plane

Also assume that the TSA correctly carries out its inspections of passengers, their boarding passes, and their photo IDs.

Mark ALL of the following concepts that are relevant to analyzing whether the airport's design achieves goals **(a)** and **(b)**. You should only consider the approach as described above. Do not consider any additional facts that you happen to know about how airport security actually works.

- ○ Code is Data and Data is Code
- ● Ensure complete mediation
- ○ Injection vulnerability
- ● TOCTTOU vulnerability
- ○ Whitelisting

Solution: The airport needs to make sure that its checks always occur, hence there are concerns regarding ensuring complete mediation. In addition, the check for the photo ID match is done separately from passengers actually boarding a plane. This introduces a Time-of-Check-to-Time-of-Use vulnerability in that once past the TSA security check, two passengers can swap boarding passes, undermining one of the security goals.

As described, there are no instances of information being treated as instructions, which both “Code is Data and Data is Code” and “Injection vulnerability” refer to. In addition, “Whitelisting” refers to only allowing a restricted, predefined set of accesses. While the photo ID check restricts access, it does not do so using a predefined list. That would only be the case if the airport had a list of “approved fliers” and only allowed them past the security check. (Actual US airport security employs *blacklisting*: the TSA’s “no-fly” list.)

There was considerable discussion on Piazza regarding just what constitutes a whitelist. The key notions concern: (1) a list created ahead of time, and not data-dependent (e.g., not created per-user), (2) analyzed to establish its safety properties, (3) analyzed to determine that it’s not overly restrictive.

(d) (12 points) Which of the following attacks can web servers protect against by sanitizing user input? **Mark ALL** that apply, even for cases where there are better ways to protect against the attack than sanitization.

- ● XSS
- ○ CSRF
- ● SQL Injection
- ○ Phishing
- ○ Drive-by Malware
- ○ Clickjacking

Solution: Sanitizing user input (more broadly, untrusted data) means transforming it in some fashion to remove or deactivate (say by quoting) elements that could be treated as instructions rather than data. This applies to XSS (for which untrusted data can be sanitized by removing `<script>` tags, for example), and to SQL Injection (for example, escaping single quotes, though this is tricky to get right).

CSRF vulnerabilities have to do with how websites structure the actions that users take. A CSRF attack looks identical to a legitimate request; thus, sanitization can't help with blocking them.

Phishing attacks and drive-by malware target browsers rather than web servers.

In Clickjacking attacks, target web servers receive requests that look identical to legitimate requests.

(e) (12 points) Which of the following are defenses against XSS vulnerabilities? **Mark ALL** that apply.

- ○ Use browsers written in a memory-safe language
- ● Use Content Security Policy headers to turn off inline scripts
- ○ Always set cookies using the "secure" flag
- ○ Prevent webpages from being framed by other domains
- ○ Make sure that browsers enforce the Same Origin Policy
- ● Use HTML escaping

Solution: XSS vulnerabilities exist due to web servers either (1) allowing users to store enriched HTML content in a manner such that other users can view it (Stored XSS), or (2) including in a reply elements derived from the URL that was sent to make the corresponding request.

CSP can defend against XSS vulnerabilities by controlling which scripts a browser will execute. In particular, one of the most powerful features of CSP in this regard is that it disables executing of "inline" scripts, meaning those that are directly included in web pages returned by the server.

HTML escaping will transform the meta-characters used in XSS into alternative representations that a browser will treat as data (e.g., `&lt;script&rt;` instead of `<script>`) rather than instructions.

XSS attacks reflect *logic* flaws (unintended functionality) rather than memory-safety vulnerabilities, so using a memory-safe language won't help.

The "secure" flag for cookies causes cookies to only be sent over a secure network channel (using HTTPS rather than HTTP). In XSS attacks, *how* the script is conveyed over the network does not matter.

Preventing webpages from being framed helps against *clickjacking* attacks, but attackers do not need to use frames in order to launch XSS attacks.

The main goal of XSS attacks is to fool browsers into executing Javascript from a target site's origin rather than from the attacker's origin (hence, "cross-site" in the name). The trickery is necessary because browsers do in fact enforce the Same Origin Policy. Thus, the SOP does not help to defend against XSS vulnerabilities—it's what inspires attackers to figure them out in the first place.

(f) (12 points) Suppose that the Acme Browser ensures that the URL displayed at the top of a given window (in the "address bar") always accurately reflects the URL of the page the browser is displaying in that window. It is not possible for any script to alter what's shown as the URL, nor to overlay text on top of the address bar.

Alice, a security-minded user, runs Acme browser. She loads a page from `hohum.com`. The address bar displays `http://hohum.com/`***`path`***, where ***`path`*** contains a bunch of text (all of which fits into the address bar, and thus is visible). Alice has no information about the trustworthiness of the `hohum.com` site.

By carefully inspecting the URL in the address bar, for which of the following Web security attacks can Alice *at least partially* **defend** herself. **Mark ALL** that apply. Do not mark an attack if Alice can only possibly *detect* that the attack happened. Only mark attacks that she can (at least sometimes) keep from happening.

- [ ] CSRF
- [✓] Clickjacking
- [✓] Phishing
- [ ] Reflected XSS
- [ ] SQL Injection
- [ ] Stored XSS

Solution: In one form of clickjacking attack, the attacker displays a legitimate page in an iframe and puts *above* it invisible elements to capture the input the user attempts to provide to the legitimate page. We saw this example in lecture with capturing keystrokes meant to be sent to Calnet authentication. Alice can defend herself against this scenario by observing that the URL shows `evil.com` rather than something like `xyz.berkeley.edu`.

Many phishing attacks display a page for the user that looks legitimate but in fact is a mock-up. We saw this in lecture for the fake Paypal account confirmation pages. Alice can defend herself against such attacks by observing that the URL shows `evil.com` rather than something like `paypal.com`.

In a CSRF attack, the cross-site request happens *automatically.* There is no opportunity for Alice to protect herself by inspecting the URL of the page she loaded.

In reflected XSS, the user clicks on a link that will reflect a script off of a target web server. However, until Alice clicks on the link (at which point it's too late), all she will see in the URL bar is the name of the page that includes the link, such as `evil.com`.

SQL injection attacks target servers rather than browsers. Alice would only observe one if for some reason an attacker decided to launch it through her browser. Even so, she would not see anything in the URL itself that would enable her to prevent the attack; she might be able to tell *after* the attack occurs that a suspicious URL was loaded.

In stored XSS, if Alice has not yet visited the part of `target.com` that includes the attacker's stored script (which is what is required for her to prevent the attack, as opposed to detect it), she will not observe any indication of the threat in the URL bar.

Problem 3 *Reasoning About Memory Safety* (58 points)

Consider the following code:

```
1 /* Copy n characters from src into dst starting at
2  * dst's start_at'th character.
3  */
4 void copy_at(char *dst, char *src, int n, int start_at) {
5     for (int i = 0; i < n; i++) {
6         dst[i + start_at] = src[i];
7     }
8 }
```

(a) (12 points) Write down a *precondition* that must hold at line 6 to ensure memory safety.

Solution:

```
Requires: src != NULL && dst != NULL &&
  0 <= i < size(src) &&
  0 <= i + start_at < size(dst)
```

Note that here `i + start_at` is referring to *C arithmetic*, not to mathematical arithmetic. The difference is important because the former includes the possibility of overflow. (The simpler constraint of `0 <= start_at` does not suffice, since `i + start_at` could overflow.)

An equivalent solution—a bit more clunky but also more explicitly clear—would be to add `... && i + start_at <= MAXINT`, with the understanding that MAXINT denotes the largest representable signed integer.

Also note that simply stating "`src` and `dst` are valid pointers" does not suffice, because a NULL pointer is a valid pointer. We mentioned in lecture that for simplicity we often *leave out* the term "…is a valid pointer" when talking about pointers, as done in the solution above.

One minor error is to misphrase the statement that the pointers must not be NULL, such as using `*src != NULL`.

Some students used `strlen()` to express size constraints on `src` and `dst`. This is not quite correct, because the function does not only apply to C strings but also arrays of bytes (which are also defined in C using `char*`); nothing in the function relies on NUL-termination of strings, and in fact the code will continue beyond a NUL if one is present. Finally, `strlen()` could itself introduce a memory-safety issue if in fact the string is not properly NUL-terminated. (One could address that fact by stating that the precondition requires `src` and `dst` to be "well-formed strings".)

In preconditions and the like, the proper way to refer to the size of the memory region pointed to by a pointer X is `size(X)`. Some students used `sizeof(X)`; this actually has a different meaning, but we allowed it because we felt it was clear the student understood the concept they wanted to convey. We similarly allowed `len(X)`, which is not a term we've used in a memory-safety context, but whose intended meaning was clear.

(b) (16 points) Write down a *precondition* that must hold when the function is called to ensure memory safety. As usual, your precondition should not unduly constrain the operation of the function.

Solution:

```
Requires: src != NULL && dst != NULL &&
  n <= size(src) &&
  0 <= start_at && 0 <= n + start_at <= size(dst)
```

We can determine the elements of the function's precondition by inspecting what we will need to support the memory-safety precondition in the previous part.

Here again we use *C arithmetic* for the term `n + start_at`. We could replace the term with `n + start_at <= MAXINT`.

A common minor mistake here was to introduce some form of "fencepost" error, such as using `n < size(src)` instead of `n <= size(src)`. Such errors slightly constrain the operation of the function.

(c) (15 points) Write down an *invariant* that always holds just **before** line 6. You can assume that the precondition you specified in part (b) is true when the function is called. For simplicity, you can omit from your invariant any terms that appear in the precondition from part (b) that will be true throughout the execution of the function.

Your invariant should allow you to establish that the precondition you stated in part (a) will then also be true.

Solution:

```
Invariant: src != NULL && dst != NULL &&
  0 <= i < n <= size(src) &&
  0 <= i + start_at < n + start_at <= size(dst)
```

Most of these terms appear in the function's precondition, and none of the variables in the precondition change during the execution of the function. So you could write this invariant as simply:

`Invariant: 0 <= i < n && 0 <= i + start_at < n + start_at`

and in fact the second clause immediately follows from the first clause (by adding `start_at` to both sides of `i < n` — which is sound because our invariant in (b) establishes that `n + start_at` does not overflow), so a solution of:

`Invariant: 0 <= i < n`

also suffices. That invariant immediately follows from the loop's initialiation and iteration condition.

Again, `i + start_at` refers to *C arithmetic*, as discussed above.

A common mistake here was to write an invariant in terms of `n` rather than `i`. Such invariants do not allow us to prove that the memory-safety precondition (from part (a)) holds.

(d) (15 points) Write down an *invariant* that always holds just **after** line 6 executes (but before the loop iterates). The same as for part (c), you can omit terms from part (b)'s precondition if they will be true throughout the execution of the function.

Your invariant should allow you to establish that your invariant in part (c) will always hold when the loop iterates.

Solution: This is the same as in part (c). Line 6 does not alter the value of any of the variables in that invariant.

A common small mistake here was to update (c)'s invariant to reflect `i` having undergone the increment listed at line 5. The phrase "just **after**" specifically indicates that nothing further has executed other than line 6, as does the term "Before the loop iterates", because loops "iterate" by applying their increment operation and then checking the guard condition.

Problem 4 ***Browser Security*** **(56 points)**

Neo has decided to build a new web browser, BerkBrowser, which mimics the design of the Chromium web browser. BerkBrowser works by splitting the browser into two separate processes:

1. A *renderer* process, which is in charge of processing a website's code and resources (HTML, CSS, Javascript, images, videos) to generate the webpage's DOM and to then display the webpage's content to the user. It is also responsible for enforcing the Same Origin Policy (SOP).
2. A *kernel* process, which is in charge of managing the browser's persistent state (cookies, bookmarks, passwords, downloads) and mediating access to the user's local file system (e.g., downloads and uploads).

When a user visits a website using BerkBrowser, the renderer process parses and displays the webpage to the user. As the website's code makes various requests, the renderer performs SOP checks and allows/denies the actions as appropriate. Under this architecture, the renderer process cannot access the user's local filesystem, and must communicate with the kernel process via a small and bug-free API in order to access files on the user's machine. If a website wants the user to upload a file, the renderer will send an API call to the kernel process, which will display a dialogue window where the user can either choose to close the window (not upload anything), or select a file to upload. If a user needs to download a file from a website, the renderer will send an API call to the kernel process, which will display another dialogue window to the user that asks if the user would like to accept or reject the download. If the user accepts, the file is downloaded into the browser's Downloads folder. If the user rejects, the file download is blocked.

For parts A and B, mark your multiple choice answer and then write a **one-sentence explanation** in the space below each question.

For part C, write your answer in the space below the question; keep your response ≤ 4 sentences.

(a) (16 points) **Mark ONE** of the following security principles that **best** describes the BerkBrowser's architecture.

- ○ Consider Human Factors
- ○ Detect If You Can't Prevent
- ○ Don't Rely On Security Through Obscurity
- [✓ ● Least Privilege Principle]
- ○ Use Fail-Safe Defaults

Solution: The renderer process is given a restricted set of capabilities; it cannot access the user's local filesystem/data without asking the kernel process for permission. This reflects a Least Privilege design.

Some students argued that the emphasis on dialog boxes in the discussion of BerkBrowser's architecture reflected Consider Human Factors. While there's some modest merit to that argument, such answers only received partial credit, since the **best** answer is the one that goes to the heart of the architecture, namely the privilege-separated design.

(b) (20 points) **Mark ALL** of the following web attacks that this architecture is primarily designed to mitigate.

○ XSS Attacks

○ SQL Injection

○ Phishing

[✓ ● Drive-by Malware]

○ CSRF

Solution: If the renderer is compromised, an attacker still cannot read/write/execute files on the user's local machine (assuming the kernel process isn't also exploited). All of the other attacks are possible if the renderer is compromised, since they do not rely on altering the browser's execution.

(c) (20 points) Suppose that Bob is using the BerkBrowser to surf the web. He visits his banking website, makes some transactions, but does not log out afterwards, so his cookie does not expire. He then navigates his browser to his favorite news website, but accidentally clicks on an ad that takes him to `evil.com`. This malicious website manages to exploit a vulnerability in the renderer process that allows `evil.com` to completely compromise and control the renderer, but not the kernel process.

Can the malicious website cause transactions to occur from Bob's bank account? If your answer is **Yes**, briefly describe how the attack would work. If your answer is **No**, explain why BerkBrowser's architecture prevents this attack.

[✓ ● **Yes**] ○ **No**

Explanation:

Solution: The renderer process is compromised, so the attacker can arbitrarily get around the Same Origin Policy protections.

Two weaker forms of attack appeared frequently in student answers. In the first, the attacker uses their position inside the browser to set up situations that fool the user in some fashion into taking an action the user doesn't intend. While such an attack will work, it misses the key notion that the attacker has the power to directly issue Web requests, and thus no need to try to trick the user into issuing them, so this answer received only partial credit.

In the second, the answer posits that the web server of Bob's bank has some vulnerability which the attacker then leverages. These answers received only a small amount of credit, as they lack the core notion that the attacker has gained a powerful vantage point by compromising the renderer process.

Problem 5 ***I don't think you heard me . . .*** **(60 points)**

Assume the code below has been compiled to use randomized stack canaries, and is run with data execution prevention (e.g., DEP), and ASLR applied to the stack segment. ASLR is not applied to other memory regions.

How might an attacker exploit a vulnerability in this code to execute the command "/bin/rm -R /home/enemy/*", deleting all of the files of user "enemy"?

```
/* DEP will be enabled! :o */
void run(char* cmd) {
  system(cmd);
}

void print_twice_the_fun(char* x) {
  printf("%s %s\n", x, x);
}

int main() {
  // random unpredictable stack canary will be used!
  char first[32];
  void (*printfn)(char*);
  char second[4];

  printfn = &print_twice_the_fun;

  gets(first);
  gets(second);

  printfn(first);
}
```

Use the following assumptions:

1. The server is on an IA-32 platform with 4-byte words (recall it's also little endian).
2. The stack is aligned at word granularity.
3. Local variables of each function are placed on the stack in the order they appear in the source code.
4. The address of the first instruction of the *run* function is 0x11111110.
5. The address of the first instruction of the *print_twice_the_fun* function is 0x11111250.
6. The address of the first instruction of the *main* function is 0x11111500.
7. A randomized stack canary protects the *main* function's RIP.
8. Data execution prevention is enabled.
9. ASLR is enabled for the stack segment.

Answer the following:

(a) (30 points) For each defense mechanism below, describe why the code is still vulnerable even using this defense.

1. Randomized stack canary

> **Solution:** Stack canaries do not protect local variables, rather they protect the RIP. In particular, this code has a function pointer that is a local variable in *main*. This function pointer will not be protected by stack canaries and is later called within *main*, meaning it can be overwritten with a buffer overflow and executed.
>
> Note that the problem asks for why the *given code* remains vulnerable; not for weaknesses in the defense in general. Thus, to earn at least partial credit an answer had to convey the relevance to the particular code. Only listing general defense weaknesses did not suffice for partial credit.
>
> Also, there was ambiguity regarding whether for this question one should analyze each defense in isolation, or issues with it given that the other two defenses were also in use. We intended the latter, but constructed a grading rubric that provided potentially full credit for either interpretation.

2. DEP

> **Solution:** DEP marks memory pages as writeable but not executable, or executable but not writable. This indicates that anything (such as the stack) which the attacker might be able to inject code into will not be executable, so instead the attacker must use existing code (which is executable already). This code is vulnerable because the function pointer can be overwritten to point to any existing code, such as the *run* function, which will be allowed to execute under DEP.

3. ASLR on the stack

> **Solution:**
>
> ASLR on the stack means memory addresses for things on the stack will be unpredictable. However, the layout of other memory regions, notably the TEXT segment, will not be randomized. This indicates that existing code, such as the function *run*, will be at deterministic and predictable locations, since they reside in the TEXT segment. Thus with this code, an attacker can overwrite the function pointer to execute the *run* function. In addition, the *run* function already accepts the *first* buffer as an input, so the chosen command string can simply be supplied as the first *gets* input, without

needing to fiddle with the address of anything on the stack.

(b) (30 points) Give inputs (for both the first and second calls to `gets`) that an attacker could provide corresponding to a successful attack. You should indicate any hex characters such as 0xff (i.e., a single byte with value 255) by writing them between vertical bars, for example: "|0xff|". So if you want to indicate the attacker inputing 3 'A's followed by two 0x8d characters, you would write "AAA|0x8d|0x8d|". You do not need to indicate the newline that terminates each line.

i. First input:

Solution: Any value you wish, as it will be irrelevant.

ii. Second input:

Solution: "aaaa|0x10|0x11|0x11|0x11|/bin/rm -R /home/enemy/*", where "aaaa" is any arbitrary 4 characters.

Note if you to set up the "/bin/rm" command in the first input, the *gets* call for the second input will *overwrite* a character when NUL-terminating, which will mess up your command passed to `system()`. So the entire exploit should happen in the second input.

It was very common for answers to overlook this NUL-termination issue. Answers that were fully correct other than for this consideration received close to full credit.