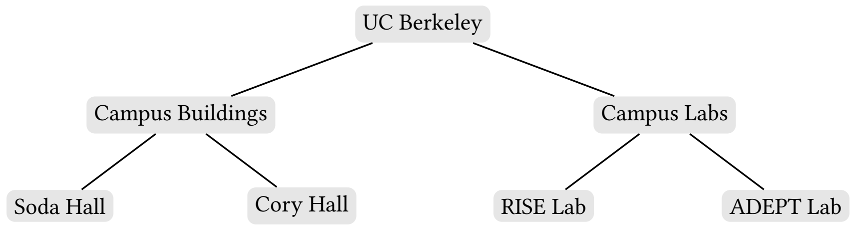
Madison & Ana
Summer 2023

CS 161 Computer Security

Discussion 7

Question 1 *RISELab Shenanigans*

Certificate authorities of UC Berkeley are organized in a hierarchy as follows:



Alice is a student in RISELab at UC Berkeley and wants to obtain a certificate for her public key. Assume that only RISELab is allowed to issue certificates to Alice.

Q1.1 Which of the following values are included in the certificate issued to Alice? Select all that apply.

■ Alice's public key

☐ Alice's private key

■ A signature on Alice's *public* key, signed by RISELab's private key

☐ A signature on Alice's *private* key, signed by RISELab's private key

☐ None of the above

Solution: This follows from the definition of certificates: they include a user's public key, and a signature on the enclosed public key, signed by the issuer (which we state in the prologue is RISELab).

Q1.2 Assume that the only public key you trust is UC Berkeley's public key. Which certificates do you need to verify in order to be sure that you have Alice's public key? Select all that apply.

- ■ Certificate for Alice
- □ Certificate for Soda Hall
- ■ Certificate for RISELab
- ■ Certificate for Campus Labs
- □ None of the above

Solution: To validate Alice's public key, we can follow our way up to our root of trust (which is UC Berkeley's public key). As such, we need certificates for Alice, RISELab, and Campus Labs.

Q1.3 RISELab issues a certificate to Alice that expires in 1 hour. Which of the following statements are true about using such a short expiration date? Select all that apply.

- ■ It mitigates attacks where Alice's private key is stolen
- □ It mitigates attacks where RISELab's private key is stolen
- □ It mitigates attacks where Campus Labs' private key is stolen
- ■ It forces Alice to renew the certificate more often
- □ None of the above

Solution: Short expiration times only mitigate the situation where Alice's private key is stolen. If RISELab's private key is compromised, the attacker can issue certificates with any expiration date, and it is up to the parent CA to revoke RISELab's certificate, not RISELab itself. The same argument applies to Campus Labs' private key.

Question 2 ***Password Storage***

Bob is trying out different methods to securely store users' login passwords for his website.

Mallory is an attacker who can do some amount of *offline* computation before she steals the passwords file, and some amount of *online* computation after stealing the passwords file.

Technical details:

- Each user has a unique username, but several users may have the same password.
- Mallory knows the list of users registered on Bob's site.
- Bob has at most 500 users using his website with passwords between 8–12 letters.
- Mallory's dictionary contains all words that are less than 13 letters. [*Clarification during exam*: Mallory's dictionary contains all possible user passwords.]
- Mallory can do N online computations and $500N$ offline computations where N is the number of words in the dictionary.
- Slow hash functions take 500 computations per hash while fast hash functions require only 1 computation.[1]

Notation:

- $\mathsf{H_S}$ and $\mathsf{H_F}$, a slow and fast hash function
- Sign, a secure signing algorithm
- uname and pwd, a user's username and password
- k, a signing key known only by Bob

If Bob decides to use signatures in his scheme, assume he will verify them when processing a log-in.

For each part below, indicate all of the things Mallory can do given the password storage scheme. Assume Mallory knows each scheme. **Unless otherwise specified, assume that she can use both offline and online computation**

Q2.1 Each user's password is stored as $\mathsf{H_F}(\mathsf{pwd} \,||\, \texttt{'Bob'})$.

■ (A) Learn whether two users have the same password with only online computation

■ (B) Learn a specific user's password

■ (C) Change a user's password without detection

■ (D) Learn every user's password

☐ (E) None of the above

☐ (F) ——

Solution: Since this is a hash function with the same salt, Mallory can do one full run through of the dictionary with online computation to learn each user's password. Additionally, there are no authenticity checks so Mallory can edit a password.

[1]Keep in mind this is much faster than a real-life slow hash function.

Q2.2 Each user's password is stored as the tuple $(\mathsf{H_S}(\mathsf{pwd} \mid\mid \texttt{'Bob'}), \mathsf{Sign}(\mathsf{k}, \mathsf{H_F}(\mathsf{pwd})))$.

■ (G) Learn whether two users have the same password with only online computation

■ (H) Learn a specific user's password

■ (I) Change a user's password without detection

■ (J) Learn every user's password

☐ (K) None of the above

☐ (L) ——

Solution: Because of the slow hash, Mallory can only longer do a full run through of the dictionary using online computation. However, she can do so using offline computation since the salt is the same for all passwords. Since the signature does not include the username, password entries can be swapped without detection.

An earlier version of the solutions incorrectly marked (A) as incorrect. However, since signatures are unsalted, an attacker can learn if two users have the same password by comparing signatures (which requires no computation).

Q2.3 Each user's password is stored as the tuple $(\mathsf{H_F}(\mathsf{pwd} \mid\mid \mathsf{uname}), \mathsf{Sign}(\mathsf{k}, \mathsf{uname} \mid\mid \mathsf{H_F}(\mathsf{pwd})))$

☐ (A) Learn whether two users have the same password with only online computation

■ (B) Learn a specific user's password

☐ (C) Change a user's password without detection

■ (D) Learn every user's password

☐ (E) None of the above

☐ (F) ——

Solution: Because the salt is now different, Mallory only has enough online computation to bruteforce a single password. However, using offline computation she can still learn all the passwords since she can bruteforce the dictionary 500 times. Since each signature is tied to a specific user and Mallory doesn't know k, she can't edit a user's password.

Q2.4 Each user's password is stored as $(\mathsf{H_S}(\mathsf{pwd} \mid\mid \mathsf{uname}), \mathsf{Sign}(\mathsf{k}, \mathsf{H_S}(\mathsf{pwd})))$

[*Clarification during exam*: The expression was missing a leading parenthesis.]

■ (G) Learn whether two users have the same password with only online computation

■ (H) Learn a specific user's password

■ (I) Change a user's password without detection

☐ (J) Learn every user's password

☐ (K) None of the above

☐ (L) ——

Solution: Mallory only has enough total computation to learn a single user's password, denoted as pwd'. She can now edit a different user's password to be this by computing $\mathsf{H_S}(\mathsf{pwd}' \mid\mid \mathsf{uname})$ and using the signature $\mathsf{Sign}(\mathsf{k}, \mathsf{H_S}(\mathsf{pwd}')))$. Note this is possible because the signature isn't bound to any specific user.

An earlier version of the solutions incorrectly marked (A) as incorrect. However, since signatures are unsalted, an attacker can learn if two users have the same password by comparing signatures (which requires no computation).

Question 3 ***Brainf[REDACTED]***

Consider the following code:

```c
void execute(char* commands, FILE *file) {
    int buf_ind = 0;
    int buf_len = 16;
    char buf[buf_len];
    size_t comm_ind = 0;
    while (commands[comm_ind]) {
        if (commands[comm_ind] == 'C') {
            buf_ind += 1;
        } else if (commands[comm_ind] == 'D') {
            buf_ind -= 1;
        } else if (commands[comm_ind] == 'E') {
            printf("%c", buf[buf_ind]);
        } else if (commands[comm_ind] == 'F') {
            printf("%x", &buf[buf_ind]);
        } else if (commands[comm_ind] == 'G') {
            fread(&buf[buf_ind], sizeof(char), 1, file);
        }
        /* assume you are provided two functions: min and max. */
        buf_ind = max(0, min(buf_len, buf_ind));
        comm_ind += 1;
    }
}
```

For this question, assume the following:

- You may use `SHELLCODE` as a 52-byte shellcode.
- Stack canaries are enabled, and all other memory safety defenses are disabled.
- If needed, you may use the standard output as `OUTPUT`, slicing it using Python syntax.
- The RIP of `execute` is located at `0xffffabcc`.
- The top of the stack is located at `0xffffffff`.
- `execute` is called from `main` with the proper arguments.

Q3.1 (4 min) Fill in the following stack diagram, assuming that the program is paused after executing **Line 6**, including the arguments of `execute` (the value in each row does not necessarily have to be four bytes long).

Stack

Solution: Stack diagram:

```
[4] File *file
[4] char* commands
[4] RIP execute
[4] SFP execute
[4] canary
[4] buf_ind
[4] buf_len
[16] buf
[4] comm_ind
```

Q3.2 (12 min) We wish to construct a series of inputs that will cause this program to execute SHELLCODE that works 100% of the time.

Provide a string input to variable `commands` (argument to `execute`):

Solution: `'C' * 16 + 'G' + 'C' * 16 + 'GC' * 4 + 'C' * 8 + 'GC' * 52`

Provide a string for the contents of the file that is passed in as the `file` argument of `execute`:

Solution: `'\xff' + '\xd8\xab\xff\xff' + SHELLCODE`

Solution:
Exploit's Concept: The `min`/`max` logic on line 19 of the function introduces an off-by-one vulnerability, allowing `buf_ind` to go one index past the end of `buf`.

Using this off-by-one and the `G` command, we can change the LSB of `buf_len` to any value of our choosing. Thus, we can increase the value of `buf_len` from 16 to 255. Now `buf_len` can further up the stack.

This allows us to now move `buf_ind` up to the `RIP of execute` and overwrite it with the address of our shellcode, and then write SHELLCODE above the `RIP of execute`.

One thing to be careful about in the previous step is that SHELLCODE cannot be placed directly after the `RIP of execute`, because this will overwrite the `commands` and `file` pointers, causing the function to not process future commands and inputs correctly.

Exploit Construction: We begin with 16 `C` commands to move `buf_ind` to the LSB of `buf_len`. Next, we include one `G` command so that we read one byte from `file` and write it to the LSB of `buf_len`. We include an `0xff` byte in `file` so that this `G` command will change `buf_len` from `0x00000010` (16) to `0x000000ff` (255). Now `buf_ind` is allowed to move anywhere between 0 and 255 (inclusive).

Taking advantage of this, we can now use 16 `C` commands to move `buf_ind` to the `RIP of execute`. Next, we will use a `G` command to read a byte from `file` into the LSB of the `RIP of execute`. We want this byte from `file` to be the LSB of the address of our SHELLCODE, which we will calculate momentarily. We then use a `C` command to move `buf_ind` to the next byte of the `RIP of execute`. We repeat three more `GC` commands to write the three remaining bytes of `RIP of execute`.

Recall that we cannot directly write the SHELLCODE right after the `RIP of execute` because we cannot overwrite the arguments of `execute`. Thus, we must write SHELLCODE after the two arguments of `execute`, meaning `buf_ind` must go up 8 more bytes. Thus, we include another 8 `C` commands. Now, we can write SHELLCODE into memory using 52 repeated `GC` commands. The address SHELLCODE will be written at is above `file`, which is 12 bytes above the `RIP of execute`. We are given that the address of the `RIP of execute` is `0xffffabcc`, so 12 bytes above this `0xffffabd8`.

Q3.3 (3 min) If ASLR is now enabled, which of the following modifications to the provided code would allow you to execute `SHELLCODE` 100% of the time? Select all that apply.

- [✓] Line 10 is replaced with `scanf("%u", &buf_ind)`.
- ☐ `jmp *esp` is located in your code at `0xdeadbeef`.
- ☐ Line 14 is replaced with `comm_ind = getchar()`.
- ☐ None of the above

Solution: We need two things to make ASLR work: we need to leak an address and we need to have a way to pause the program. We can pause the program with `scanf` and `getchar`. However, `getchar` gets rid of the line that leaks the address. `jmp *esp`, while it would normally work, overwrites the file pointer, which is required during the `fread` calls.