Introduction to Cybersecurity Elisabeth Oswald

System Security

2023

Overview

Some exercises are based on paper and pencil. Unless you use a tablet (with a pencil) you must bring actual paper and pencil along.

For some parts of this lab you need to program in Python, and we expect that you have a reasonable level of proficiency in Python already. We also expect that you have Python installed on your system. You will also need the Cryptography library as well as PyCryptoDome.

This lab sheet covers the contents of both weeks that pertain to System Security (there is a separate work sheet on password security). We do not expect that you can finish all exercises within the (two) 3hr lab session(s). We also do not require that you do all exercises; but some are necessary to solve your Moodle quiz.

What you should get out of this.

System security is a vast area and we can, at best, dip into some topics that are of interest these days. To support your learning, we supply some examples that are sufficiently hands-on without requiring installation of complex emulators etc.

Assessment

There is no required submission for this lab, but doing the tasks will enable you to answer questions in a small Moodle quiz which is due at the end of next week.

Paper and Pencil Exercises

Exercise 1: Bell-La Padula/Biba

Consider this toy example of a an access control matrix: the first column contains the restriction levels, the second column contain a short list of employees on the resp. level, and the third column has a description of files at the resp. level.

Level	Employees	Files marked as
Top Secret	Alice, Adam	private
Secret	Bob, Barbara	for sharing within subunits of the organisation
Confidential	Carol, Craig	for sharing across the entire organisation
Unclassified	Dave, Dora	for unrestricted sharing (also outside the organisation)

In the context of the Bell-LaPadula model: can Adam write to files that are shared with members outside of the organisation? Can Carol read documents that are shared within subunits of the organisation? Try and answer these questions following the Bell-LaPadula logic for access control.

In the context of the Biba model: can Barbara modify private files? Can Dave view files marked as for sharing across the entire organisation?

Then make up some more statements and challenge your neighbour (or any fellow student)!

Exercise 2: Fault Attacks

Consider the `match()` function from the lecture. We argued that this function is not secure with respect to fault attacks. Find at least three different ways to rewrite this function (you must maintain its functionality) that change its behaviour with respect to fault attacks. Discuss for each of your three new variations how fault attacks could still apply.

```
bool match(char* C,char* P,int Clen,int Plen)
{
 bool value=true;
 if ( Clen != Plen ) { value=false; }
 for (int i=0; i<Plen && value==true; i++)
 {
     if ( C[i] != P[i] ) { value=false; }
 }
```

```
  return value;
}
```

Exercise 3: Single trace attack(s) on PKC implementations

In the context of public key cryptography (this includes encryption, key exchange, signatures, etc. that we have studied in this course), find as many systems (practical or Vanilla schemes) as you can where a power attack **using a single input only** could break the corresponding scheme. Argue, for each scheme how you would apply the power attack, what it would recover, and how this would lead to the break of the scheme.

Programming Exercises

Exercise 4: Timing Attack

A very simple, yet devastating, way to attack verification oracles in practice is to use timing information. In this task, you will find out why such timing information might exist, and how to exploit it.

Check out the Python script "timing.py" which is supplied alongside this lab sheet. It implements a simple verification function called `match` via compairing a supplied string to a "secret string" in a straightforward manner. This kind of verification is often used to test passwords, and PINs, but it is also equally used in the verification of MAC values (i.e. in the context of canonical verification where a tag is computed and then compared to another, supplied, tag value).

The `match` function simply does this verification by first checking if the supplied value is of correct length, and if this checks out, then the function compares the supplied with the secret value, character by character. As soon as a mismatch is found, the function aborts. This is a very natural way to implement such a comparison: clearly a difference in a single character suffices to reject the supplied value, thus it is "efficient" to abort as soon as a mismatch is found. Similarly, if the length check doesn't succeed, there is not point in checking the characters, thus an early return (as happens in the function) makes sense from an efficiency point of view.

However, the problem that we introduce by only caring about efficiency is that we create a timing side channel: the longer the match function takes to complete the more things must have been correct. This can be used to develop attacks that are much more efficient than a brute force search.

This task is about developing such an attack. We will do this in two stages: first you will use the timing side channel to determine the length of the unknown secret value, and secondly, you will use the timing information to reveal the secret value itself. In order to make things slightly easier for you, I have added a little time delay in the inner loop, which exaggerates the timing differences. In a real world scenario, one needs a better timer, or one needs to do repeat measurements (I discuss such aspects in the Security Engineering course). There is also no maximum number of guesses that you can send to the `match` function.

Derive the length of the secret string

Write a function with signature `int guess_length()` which returns an integer value representing the length of the secret string. I give you a bit of a helping hand by providing a line of code that creates a random lower case ASCII string of length `len_value`, and by giving you the hint that it is helpful to store the timings that you get from the `match` function in an array. The array entry that corresponds to the longest execution time indicates the length of the unknown secret value.

```
def guess_length():
    duration = [] # array to hold the different amounts of time used to
         check guesses for the secret string
    rvalue = ''.join(random.choice(string.ascii_lowercase) for j in
        range(len_value)) # random guess of length len_value

    # insert your code here that automates checking of rvalues of
        different lengths using the match function

    return duration.index(max(duration))+1)
```

You can use the supplied script `timing.py` to write/test the `guess_length()` function. Once you are sure it works, you are ready to use it in the respective Moodle quiz question.

Derive the secret string

Using the same technique as in step before, you can also recover the secret string itself. This means that you can check, character by character, which of the “allowable” characters of the alphabet lead to the longest execution time: assuming you have correctly recovered the string up to l characters, you can find the $l+1$th character by submitting test strings of length $l+1$, whereby the $l+1$th character varies (and the other characters are the ones that you have already successfully recovered).

For instance, assume that you already know that the secret string starts with “st”, then you would submit the guesses “sta”, “stb”, “stc”, etc. to the match function. The guess that takes longest to evaluate will indicate the correct guess.

Write a function called `recover_string(length)`, which takes as input the length of the string that should be recovered (you can reuse the code from the previous step), and outputs a string (the recovered secret value). You can restrict yourself to strings that only contain lowercase ASCII characters.

```python
def recover_string(length):
    alphabet = string.ascii_lowercase
    likely_guess = ""
    for l in range(1, length+1):
        duration = [] # array to hold the different amounts of time
            used to check guesses for the secret string

        # insert your code here that determines the secret string

    return likely_guess
```

Consider the attack complexity

Now that you have your timing enhanced attack vector, consider the complexity of your final attack. Assuming that the secret string is increased linearly, how does the attack complexity (in terms of time and memory) increase? How does this compare to a brute-force search without extra information?