University College London
Department of Computer Science
0133: Distributed Systems and Security (Brad Karp)
Individual Coursework 2: Topics in Security
Distributed: 28th November 2023; Due: 15th December 2023, 4 PM (via GitHub)

Answer all five of the following problems. All submissions must be in the form of a PDF file, whether typeset or a clear scan of a handwritten solution. If you do write by hand, please ensure your answers are legible. Please show all work! We cannot award credit for correct answers if their complete derivation isn't shown. Please state clearly all assumptions you make while solving a problem. This coursework is worth 15% of your final grade for 0133.

Please monitor the 0133 Ed site during the period between now and the due date for the coursework. Any announcements (*e.g.*, helpful tips on how to work around unexpected problems encountered by others) will be posted there.

Hand-in instructions: All submissions for CW2 are to be made electronically via GitHub. The deadline is 4 PM GMT on 15th December 2023.

Before you follow the instructions below to create a repository for your CW2 submission, you MUST first complete the 0133 grading server registration process.

If you have not yet done so, **STOP NOW,** find the email you received with the subject line "your 0133 grading server token," retrieve the instructions document at the link in that email, follow those instructions, and only thereafter proceed with the instructions below for creating your CW2 repo.

There are three steps to submitting, and you must complete all three steps fully for your submission to be accepted. First, create a private repository for your submission by visiting the link:

`https://classroom.github.com/a/5TJ7_IlB`

Second, check out a copy of the repository. The repository will initially be empty; there are no files provided for use in CW2, and your submission will consist only of the PDF file for your solutions to the problems set below in this handout.

Third, after completing your solutions to the problems, add the PDF file containing your solutions to CW2 to your repository, and push all your changes to GitHub.

We will mark only the *first push you make* of your PDF solution document to the GitHub server, and use the corresponding timestamp on the GitHub server as the time of your CW2 submission.

Late submission of this coursework will be penalized according to the standard UCL policy for late work. In brief, that policy is as follows:

- If your total mark on this coursework before any late submission penalty is applied is between 50-100, late submission penalties are:
  - Less than 2 working days late: deduction of 10, with mininum resulting score of 50.
  - 2-5 working days late: mark of 50.
  - More than 5 working days late: mark of 1.
- If your total mark on this coursework before any late submission penalty is applied is between 1-49, late submission penalties are:
  - Less than 2 working days late: no penalty.

  - 2-5 working days late: no penalty.
  - More than 5 working days late: mark of 1.

Collaboration: Collaboration is *not permitted* on this problem set; you may not discuss the problems or their solutions with anyone else (whether or not the other person is taking the class), apart from the instructor and teaching assistant. All work you submit must be your own.

Materials used and citation: You may of course refer to all lecture notes and readings for 0133 as you prepare your solutions to the questions in this coursework. You may use other reference materials (*e.g.*, textbooks, papers, or other technical material found on the Internet), with two important exceptions. First, you *may not* look at nor base your solution on materials, whether online or offline, directed specifically toward solving a problem found on this coursework (or a problem very close to it). Second, you *may not* use material from question-and-answer online forum sites—neither by asking questions related to this coursework yourself nor by searching for related questions others have previously posted. *The use of materials produced by others directed toward answering a question on this coursework, or the asking of questions related to this coursework on online forums or technical discussion web sites of any kind (other than the class Ed site), or using material from these technical discussion web sites (other than the class Ed site) is strictly prohibited, and will be treated as academic dishonesty by the instructor, resulting in the awarding of zero marks and referral for disciplinary action by UCL.*

Finally, note that you MUST fully and clearly cite all materials other than the class lecture notes that you have referred to while preparing your solutions. *Not citing a source will be treated as academic dishonesty by the instructor, and dealt with as above.*

1. **The RSA Public-Key Cryptosystem**

   Suppose you are given an efficient algorithm, `RSA-Break()`, that, for a given RSA public key (n, e), is able to decrypt 1% of the set of all possible ciphertexts encrypted with that key (without knowledge of the corresponding private key). That is, for a single given ciphertext, `RSA-Break()` will *deterministically* either always produce the entirety of the corresponding cleartext, or return a failure result. By "1% of the set of all possible ciphertexts," we mean that for 1% of the set of all possible ciphertexts, `RSA-Break()` always succeeds, whereas for the other 99% of the set of all possible ciphertexts, `RSA-Break()` always fails. Describe an efficient algorithm that uses `RSA-Break()` as a building block, and can decrypt *any* message.

   [10 marks]

2. **Format String Vulnerabilities**

   You and your friend discover a format string vulnerability in a popular server, and decide to write an exploit for it that will make the server crash. An excerpt of the C source code for the function containing the vulnerability follows:

```
int vulnerable(void)
{
  char userinput[1024];
  ...
  sprintf(outstr, userinput);
  ...
}
```

Both `outstr` and `userinput` are of type `char *`. Each of these two pointers is four bytes in length. `userinput` is a string that the server reads directly from a network socket (*i.e.,* the content of the string will be taken unmodified from within a request you can send the server). Assume that `outstr` is extremely large, such that you can be certain you won't overflow it, no matter how many characters are printed by `sprintf()` during the processing of your exploit.

Your friend analyzes the behavior of the server, and determines the following facts:

- When the vulnerable server software is run under the version of Linux used on the server you wish to target with your exploit, inside the function `vulnerable()`, the return address for resuming execution in `vulnerable()`'s caller is stored on the stack at memory address `0xbfff8218`.
- When `sprintf()` begins processing the format string `userinput`, its "next argument to print" pointer points to a memory location that is exactly 48 bytes lower in memory than the location of the buffer `userinput`.

Assume that memory address `0xdeadbeef` is unmapped in the server process, so that if execution at this address is attempted, the server will crash.

Design and supply the exact format string that, when placed verbatim by the server into `userinput`, will cause the server to crash by attempting execution at address `0xdeadbeef` when the function `vulnerable()` returns. Provide diagrams of the stack showing the steps in the execution of your format string exploit.

N.B. that you should *not* use a buffer overflow vulnerability in your answer—you must use a format string vulnerability only.

[10 marks]

3. **TAOS Applied to SSL/TLS and NFS**

In the assigned reading on TAOS, the authors describe how to use logical statements to model authentication in distributed systems formally.

(a) Suppose that Alice uses a web browser on her desktop computer to place an order with `amazon.com`. The communication between her browser and `amazon.com`'s web server uses SSL/TLS 3.0, with RSA authentication (as described in class during the SSL/TLS lecture).

Using the notation and axioms defined in Section 2.1 of the TAOS paper, write out a formal derivation that proves that Alice's web browser can trust that a response RSP received over the SSL/TLS channel from `amazon.com`'s SSL/TLS web server was sent by the real company *Amazon.com, Inc.* (Assume that *Amazon.com, Inc.* has registered a public key with an SSL/TLS certification authority trusted by Alice's web browser, in the usual way.) Your derivation should conclude with the statement:

Amazon.com, Inc. **says** *RSP*

[7 marks]

(b) In a completely different context, consider an NFS deployment enhanced to use TAOS for user authentication, where each RPC request from a client workstation is authenticated by the NFS server in the exact fashion illustrated in Section 2.2 of the assigned TAOS reading.

Suppose an attacker, Mallet, wishes to gain access to Bob's files stored on the NFS server. To do so, Mallet installs an unmodified but outdated version of the OS on Bob's client workstation.

Mallet happens to know that this version of the OS contains a vulnerability that will allow him to inject malicious code into the OS kernel while Bob is logged in and accessing his files over NFS. Mallet can thus inject malicious code into the kernel that sends Mallet a copy of the data for Bob's files (since the file data are fully visible to the kernel while Bob is logged in and using his files).

Assume that TAOS is willing to boot any unmodified, released version of the OS on a workstation, including outdated ones.

Even so, TAOS provides adequate mechanisms to prevent Mallet's attack from succeeding. Explain exactly what mechanisms TAOS provides to thwart this attack, and how Bob or the NFS server administrator should configure TAOS's authentication functionality on the NFS server to do so.

Hint: as stated in the assigned reading on TAOS, the boot firmware of a workstation can accurately determine the OS revision it is booting ...

[3 marks]

4. **Kerberos**

Kerberos Version 4 (the version of the protocol in the paper assigned for class) uses authenticators to protect against replay attacks.

Suppose that all nodes in a Kerberos realm have properly synchronized clocks, and that the clock synchronization system is secure against an adversary's manipulation of any node's clock.

The MIT Athena Kerberos deployment honored a Kerberos authenticator for 5 minutes beyond the timestamp within the authenticator. An eavesdropper could thus replay an overheard ticket and authenticator for five minutes, given that servers in this deployment didn't cache past authenticators to ensure they weren't reused.

(a) Describe an alternate, bidirectional protocol between server and client to replace the authenticator that prevents such replay attacks, and requires no new keys beyond those already in use in the Kerberos system. Your design must not require caching more than a constant amount of state per Kerberos principal on the server.

[4 marks]

(b) Read the Appendix of the assigned reading on Kerberos, which describes the authors' Kerberized NFS implementation. In this design, NFS does not include a Kerberos ticket and authenticator in every NFS request from the client workstation to the server. When *does* the user's client workstation authenticate the user to the NFS server in this design, and how?

[3 marks]

(c) TAOS allows delegation: a user may delegate authority to a workstation acting on his behalf, and that workstation may in turn delegate authority to another workstation, acting on behalf of the same user. Does Kerberos support this kind of delegation? That is, can a user on workstation *A* obtain a ticket granting access to a service on some server, and forward that ticket as-is to another workstation, for use by that other workstation when requesting services from the same server? Why or why not?

[3 marks]

5. **SSL**

When SSL 3.0 uses RSA authentication during the handshake at the start of the SSL connection, it doesn't provide forward secrecy. Describe a modification to the SSL 3.0 handshake, still *only using RSA during the handshake*—you may *not* introduce additional ciphers, and that means you **may not use Diffie-Hellman key exchange!**—that will provide forward secrecy. Please show a timeline for your modified SSL handshake (of the form of the one given in lecture), indicating the interleaving of messages sent and received by the client and server, the contents of each message, and showing when the client and server execute any other operations required to implement your modification correctly. What, if any, are the added costs of your solution over those of the "basic" RSA SSL handshake? *Hint: consider what fundamental property the key the server uses to decrypt the pre-master secret must have for forward secrecy to hold.*

[10 marks]

Problem set total: 50 marks