Homework Solutions: Lectures 6 - 11

Lecture 6

(P&P #2) **Test a microprocessor for undocumented features.** There are (at least) two goals: (1) determine whether any of the opcodes behave unexpectedly with certain combinations of opcodes, and (2) determine whether there are any undocumented opcodes.

For (1), it is clear that it is impossible to try all possible operands, e.g. a multiply command with two 64 bit operands has 2^{128} possible operand combinations. First identify a small number of classes of operands — constants, registers, direct memory addresses, indirect memory addresses — so that it is possible to try all combinations of classes, e.g. (const, const), (constant, register), etc. For each class, write code to generate random operands of that type. Now, for each opcode and each possible combination of operand classes, generate a large number of random operands and test that the opcode behaves as expected for each set of operands.

For (2), look for unused opcodes and attempt to execute them with different combinations of operands generated using the classes from part (1)

(P&P #3) **Test a computer program for undocumented features.** In this case, we need to see if there are inputs to the program that result in unexpected behavior. In that sense, it is similar to testing the operands of a microprocessor. However, whereas for a microprocessor we may be able to categorize the possible operands, a computer program may have many more possible inputs. The first step is to identify all (or as many as possible) points at which the program accepts input and to generate random inputs that can be used to test the program. Note, too, that the behavior of the program may depend on a *combinations* of separate inputs, making it extremely unlikely that a tester could produce the combination needed to trigger the undocumented behavior. An analyst could resort to examining source code or reverse engineering the binary, but this is a tedious and time consuming undertaking.

(P&P #4) **Automated testing for trapdoors**. No. Trapdoors can be extremely subtle and may be indistinguishable from legitimate authentication procedures.

Lecture 8

(P&P # 11) **Overmodularized Code.** Modules could be grouped into functional units — the descriptions of the functional units would allow a reader to get a higher-level view of how the code functions.

(P&P #12) **Testing a hash coded list**. Although we know the table size, we are not told the hashing algorithm that is used. We need to be sure that the code handles hash collisions correctly. To do this, we should generate a large number of items (greater

than the square root of the table size — why?) and check that they can be retrieved from the table correctly. If hash collisions are not handled correctly, data could be exposed to unauthorized users.

(P&P #15) **Processor that uses encrypted code and data.** Both data and code would be unreadable to an attacker who gained access to disk or memory, so long as the attacker did not know the "key" needed to decrypt the code and data. However, there must be *some* way for the processor to process unencrypted input and produce unencrypted output, or it would be useless — no human would be able to program it or use the results of a program! Management of the secret "keys" needed to encrypt and decrypt data would be difficult. Would each processor get a different key? Then every copy of the software must be encrypted with a different key, as well. What if all the processors use a common key? Then an adversary will expend great effort to get hold of that key, and if they do, will be able to read code and data with impunity.

Perfect virus detection. This question is either profound or silly. If a perfect detector *D* exists, then Program *V* is not a virus, so it is not detected; if *D* is not a perfect detector, then *V* is a virus which is not detected. So, in either case, *V* is not detected by *D*. But it seems that by this reasoning, if a perfect detector *D* existed, there would be no viruses left to detect, which sounds like a good result to me!

Picking up a USB stick. Worm or virus, which could install a rootkit, spyware, or other malicious software.

Lecture 10

1. Classified systems on an isolated network.
2. Sensitive jobs completed and programs and data removed before the machine is allowed to be used for general research jobs.
3. Customer record — sensitivity increases if the user decides to save their credit card information.
4. Wrong! The code must be protected from modification while in memory.
5. Address of the fence register can be added to all memory addresses to provide true physical memory addresses.
6. If the base/bounds registers are part of the context switch, it is possible for the memory of each concurrent process to be protected from other processes.
7. There are legitimate reasons for code to modify itself (see self-modifying code on Wikipedia). Also, it is possible for data to be read-only for a given process.
8. As seen in the previous problem, it can still be the case that areas of code will need different protections (some writable, some not) and similarly, some portions of data may be writable while others are not. Tagging would still be useful in this case, although perhaps with more overhead than with other approaches such as segmentation.
9. Other possible accesses: *delete* (allows user to delete data, not just *write*), *administer* (the ability to grant rights to other users), etc.

10. No. The users may have different permissions since the segment must be in both users' segment translation tables and different permissions can be applied by the OS at the time of access.
11. If the physical address was mapped to a different process while the device input was occurring, the data could be exposed to a process that should not have access.
12. The directory lists the objects to which the user has access and the granted access rights. Allowing a user to change his or her own directory would allow them to set their own access rights.
13. Generally speaking, you do not want other users to know what objects you have access to. This information would be helpful when choosing targets for an attack.
14. Describe each access control mechanism…
    a. Easy to determine a given subject's access to a specific object, but difficult to determine all users with access to a specific object (have to look at every subject's list); easy to add new subject access (just add object to subject's list); easy to delete a given subject's access to a specific object; difficult to create a new object to which all subjects have access (have to modify list for every subject).
    b. Easy to determine a given subject's access to a specific object, but difficult to determine subject's access to all objects (have to look at every object's list); easy to add new subject access (just add subject to object's list); easy to delete a given subject's access to a specific object; easy to create a new object to which all subjects have access.
    c. All are easy.
    d. Easy to determine a given subject's access to a specific object (subject presents ticket), but difficult to determine all subjects with access to a given object; easy to add access to a new subject (issue ticket to new subject); easy to delete access by a given subject (invalidate ticket); a pain to add an object so that all subjects have access (have to issue ticket to every subject).
15. Leave a placeholder object, flagged as "deleted", and only modify subject tables if they attempt to access the deleted object.
16. The ability to prevent writing to files protects the integrity of the data in the files, ensuring that only authorized users can modify sensitive data.
17. A process that spawns a sub-process should be able to transfer capabilities to the sub-process.
18. The ability to transfer a capability should be specifically granted — that is, the ticket that A gives to B should not be valid for capability transfer. More generally, the ticket could include a counter initialized to the number of permitted transfers; whenever a transfer occurs, the OS decrements the counter by one.
19. Physical separation is inconvenient — it is difficult to transfer data between disconnected systems and impossible to use normal communications channels such as email. It may also be wasteful in that if the isolated resources are underutilized, it is difficult or impossible to use them to process jobs from outside the protected domain. Temporal separation is inconvenient for users who must wait for sensitive jobs to complete before compute resources are available to them.

20. The process that requested I/O may not be active at the time data is returned from the I/O request. The OS needs to ensure that data is not exposed to unauthorized processes.
21. The key to this problem is the fact that only 20 users will be active at any time. Each user's directory (list of objects and access rights) should be stored on disk. When a user becomes "active," their directory is used to populate one row of a 20-by-50,000 Access Control Matrix. With twenty active users, the entire ACM only requires 1,000,000 entries, and assuming all access rights can be represented with a single bytes (i.e. only 256 different combinations of rights), the entire table is just 1 Mb in size. Using an ACM makes all access checks straight-forward. Any changes to an active user's row in the ACM will cause the user's directory to be updated on disk.

Intel Memory Protection / Linux Memory Protection. Short answer: current Intel chips provide segmentation and paging. However, Linux does not use segmentation — each process sees a single linear address space.

Lecture 11

The Birthday Problem

$$q(N) = (1 - 1/d) (1 - 2/d) \ldots (1 - (N-1)/d)$$
$$p(N) = 1 - q(N)$$

$$p(N) \approx 1 - \exp(-1/d) \exp(-2/d) \ldots \exp(-(N-1)/d)$$

$$= 1 - \exp(-(1 + 2 + \ldots + (N-1)) / d)$$

$$= 1 - \exp(-(N-1)N / (2d))$$

$$\exp((N-1)N / (2d)) \approx 1 - p$$

$$-(N-1)N \approx 2d \ln(1 - p)$$

$$N^2 - N \approx -2d \ln(1 - p)$$

$$N \approx \sqrt{(-2d \ln(1 - p))}$$

With $p = 1/2$, we get

$$N \approx 1.177 \sqrt{d}$$

A Simple Hash

If the message is short and non-random, finding a pre-image may be possible. For example, if the a_i are ascii characters, N is large, and the message is known to be three

characters long, then it is not un-likely that we could recover the message “RUN” from the hash 245.

Finding collisions is trivial. If I know a message M has hash h, then any message

$$M \,||\, b_1 \,||\, b_2 \,||\, \ldots \,||\, b_n$$

with the sum of the b_i's a multiple of N will also have hash h.

Different Hashes

You should say something about the sponge construction for SHA-3 versus the word-based round function for the other SHA algorithms. Also, the fundamental functions of SHA-3 are not simple binary functions applied many times but are more complex modifications and rearrangements of the state.

Uses of MD5

The point here is that people have been able to construct “strong collisions” using MD5, that is, two separate files that hash to the same thing. No one has demonstrated the ability to, for example, compute pre-images.

For public key certificates, we saw in class that this is a bad use of MD5. Lenstra et al. demonstrated the ability to create fraudulent X.509 certificates.

For password hashes, MD5 isn't so bad. The ability to create “strong collisions” doesn't help an attacker to find a password (a weak collision, however, would allow an attacker to find an equivalent password — one that hashes to the same thing as a legitimate password).

The security of HMAC depends on the security of the hash. Since there are collision problems with MD5, it should not be used in such a sensitive application.