CS 166 ASSIGNMENT 3 SOLUTION

Overview

Marks on question:

- No mark: 1-part short answer question (the key point(s) for explanation is underlined)
- !: 1 part auto-graded question (answer in **bold**)
- +: n-parts question with autograding part and explanation (justify or show calculation) part. The answer for autograding part is in **bold**, the key point(s) for explanation is underlined.

Basic rules of grading: Full points if your explanation/equation was correct, even if the answer is wrong (because points already deducted in the fill-in-blank part). And you get 0.5 (on whole question, not each sub-question) as long as you wrote something.

Week 5 - Software Flaws & Malware (Q1 - Q8)

!Q1: Malware timeline (L10 P7 - 10)

All correct. Summarized on L11 P2.

!Q2: Malware detection (L10 P11 - 14)

See quiz answers. Summarized on L11 P3.

!Q3: Evade signature detection (L10 P15 - 17)

3.1 See quiz answers.

3.2 Summarized on L11 P4. Note that the order of the options may be different from what appears on your assignment.

- (True) When encrypting the malware, there's no need for attackers to choose a strong cipher.
  Since it's not for confidentiality; it's trying to avoid or disguise the common signatures.
- (False) There's no way to detect an encrypted worm since we can't find common signature.
  Since the worm need to be decrypted to work, there will be a decryptor. And we can detect such worms by finding a signature in the decryptor (more-or-less standard signature detection).
- (False) We can run polymorphic worm in a sandbox (emulator) and then search for decryptor's signature.
  We run the polymorphic worm in a sandbox so it can decrypt itself. Therefore, after it got decrypted, we can see the original code and do a normal signature detection (i.e., search for signature of the whole malware, not for the decryptor only).
- (False) We can easily detect metamorphic worm since they are not encrypted.
  Although metamorphic worms are not encrypted, they are mutated. And detecting such worm is a difficult research problem.

+Q4: Canary to detect buffer overflow (L9 P15)

I. When the canary is overridden, it means the buffer overflowed and the return address may also be overridden.

II. **True**, Microsoft's implantation is flawed. Attacker can also specify the handler to execute the "evil code". This actually makes attackers' life easier since they don't need to find the appropriate address to execute their evil code.

Q5: Defending stack-based buffer overflow attacks (L9 P15 & P16)

I. (Mention at least 2) Use non-executable stack; use ASLR; use safe languages (Java, C#); use safer C functions

II. (Explain the 2 ways mentioned in I.)

- Use non-executable stack so Trudy can't execute her evil code by overflowing the stack to redirect the return address.
- Use ASLR to randomize place where code loaded in memory so Trudy can't find the address to redirect to.
- Use safe languages/safer C functions that will throw exception and crash the program if buffer overflowed.

+Q6: Incomplete mediation (L9 P17 & P18)

I. If input is not validated, it is called incomplete mediation.

II. **False**, need also validate on server side since attacker can change the input after client side verifies the input. For the example, you can make up your own, or just use the one given in L9 P18.

Q7: Race condition attack example (L9 P19)

I. Race condition attack. Since reading (step 2) & writing (step 5) data are 2 “stages” separated by step 3 & 4.

II. Various attack possible, and the curial step always happens between step 2 (reading data) and step 5 (writing data).
For example, in step 1) we can put 2 cards - card A has $500 while card B has $1 - on top of each other so they are inserted together. Make sure the machine reads card A in step 2). That is, x = 500. Then in step 3) insert a $5 bill so machine stores y = 5.
Before step 4 (press ENTER), take out card A but leave card B. Then in step 5 (after pressing ENTER), machine will write x + y = $505 to card B! Card A still has $500, and card B now has $505 although you only added $5!

Q8: Heap-based buffer overflow example (L9 P11)

I. The following is printed: BEFORE: buf2 = 22222222 AFTER: buf2 = 11122222

II. The content of buf2 changed since buf1 overflowed and the last three ‘1’s written to buf1 overrode the first 3 chars of buf2.

Week 6 - Other Attacks on Software (Q8 - Q15)

Q9: Mitigating an SRE Attack (L11 P13 - 16)

I. (Mention at least 3) Anti-disassembly; anti-debugging; tamper-resistance; code obfuscation

II. (Explain the 3 ways mentioned in I.)

- Anti-disassembly to confuse static view of code. E.g., encrypt object code or put some junk assembly instructions, etc.
- Anti-debugging to confuse dynamic view of code. E.g., check IsDebuggerPresent or monitor use of debug registers, etc.
- Tamper-resistance to make patching harder by detecting if the code is being changed through hash value.
- Code obfuscation means making code hard to understand so it takes the attacker more time to analyze.

Note that writing “anti-D’s” is consider as wrong answer as explained in class.

Q10: Linearization attack based on timing (L12 P7 - 9)

I. **False**, the code can't prevent a linearization attack based on timing. There is still time difference between correct and incorrect case: if a character is wrong, flag = false will be executed, which may take time. In this case, an incorrect serial number takes more time to check than a correct one.

II. If doesn't use linearization attack, need $32^8 / 2 = 2^{39}$ tries on average (note that $32^8 / 2 \neq 32^7$);
if use linearization attack based on time, need $32 * 8 / 2 = 2^7$ tries on average.

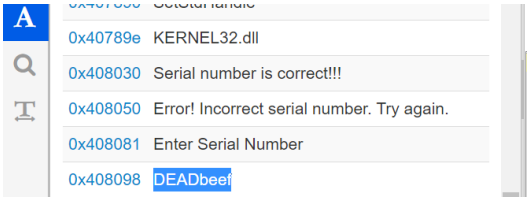
!Q11 (Extra Credit): Try it yourself - linearization attack based on timing (L12 P7 - 9)

Recover one character at a time by finding the one that takes the longest time.
10 * 6 / 2 = 30 tries on average.
The correct number is: **315972**

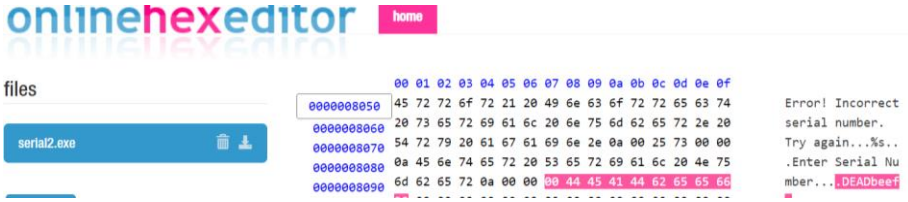
+Q12 (Extra Credit): Try it yourself - simple SRE (L11 P10 - 12)

You can use whatever disassembler and hex editor you like. I used the online ones I showed in class.

I. Once disassembled, check strings around "Enter Serial Number". The serial number is **DEADbeef**.



Or in the hex editor, look for "correct", "enter serial number", etc. and you should also be able to find the serial number as well.



II. Similar to what we did in class.
We want the program to jump to the address of "Serial number is correct" for all inputs. As we can see, the address is 0x408030. Checking the assembly code, we see loc_00401075 will push 0x408030 and print it. In order to jump to loc_00401075, the zero flag needs to be 0 (je means jump if zero flag = 0). Therefore, change test %eax, %eax to xor %eax, %eax. Note that the opcode for test %eax, %eax is 85 c0, and the opcode for xor %eax, %eax is 31 c0 or 33 c0.

Finally, open any hex editor you like, change 85 c0 to 31 c0 (or 33 c0) and save the new .exe.

```
.text:0040105d  83c418          add $0x18,%esp
.text:00401060  85c0            test %eax,%eax
.text:00401062  7411            je loc_00401075
.text:00401064  6850804000      push $0x408050
.text:00401069  e871000000      call func_004010df
.text:0040106e  83c404          add $0x4,%esp
.text:00401071  83c428          add $0x28,%esp
.text:00401074  c3              ret
.text:00401075
.text:00401075                  loc_00401075:
.text:00401075  6830804000      push $0x408030
.text:0040107a  e860000000      call func_004010df
.text:0040107f  83c404          add $0x4,%esp
.text:00401082  83c428          add $0x28,%esp
.text:00401085  c3              ret
```

change to xor,
so zero flag = 0 always
and go to loc_00401075

want always jump to here

After patching, the .exe will accept any input as correct serial number:

```
C:\Users\jwxzz>C:\Users\jwxzz\Desktop\serial2_v2.exe

Enter Serial Number
1
Serial number is correct!!!

C:\Users\jwxzz>C:\Users\jwxzz\Desktop\serial2_v2.exe

Enter Serial Number
adfhalre
Serial number is correct!!!
```