CS155 Homework 1

Leo Martel SUNet ID: lmartel

April 30, 2015

Problem 1

(a) A standard buffer overflow attack overwrites the return address stored in the stack frame during the execution of the function, so that upon return, execution resumes in another location (of the attacker's choosing). This would be detected by the "backup" system, because the overwritten value would not match the stored value, so the program would exit before running the attacker's code.

To defeat this system, an attack would need to A) guess or find the random location L and B) successfully overwrite that location as well as the stack location. This is clearly a strictly harder problem.

(b) Something like the code below, which calls a function pointer stored on the stack, is still vulnerable to a "standard" buffer overflow. An attacker can overflow 'ours' and overwrite 'fp' in 'foo'. Since 'fp' is not backed up and verified, the attacker can successfully redirect the function call.

```
int allgood(void){
  return 0;
}


int foo(char *src, int(*fp)(void)){
  char ours[1024];
  strcpy(src, ours);
  return fp();
}


int main(int argc, char *argv[]){
  return foo(argv[1], &allgood);
}
```

Problem 2

$0x40044444$	ret from Frag3
$0x40002948$	ret from Frag2
$0x41000000$	pop into ebx
$0x40002700$	ret from Frag1
$0x40000000$	pop into eax (∗)
$0x40002000$	ret-addr
$0xDEADBEEF$	SFP (never used)
string buffer, contents irrelevant	

Explanation:

After the buffer overflow, the function returns. The base pointer is moved into the stack pointer, so the bottom of the stack is now (∗). The SFP is popped into the base pointer, but this will never be used so it doesn't matter. Execution continues at the ret-addr, which we've overwritten to point to Fragment 1. This pops one word into eax, then a second word into eip, jumping into Frag 2. Frag 2 pops a word into ebp and another into eip, jumping into Frag 3. Frag 3 does the desired memory write, then pops and jumps (to our goal location).

Problem 3

This explanation assumes that the attacker may choose the size, and write to the subsequently allocated memory. This could happen in a situation where (for example) an iOS developer stores a user-generated string on the heap.

An attacker can overflow memsize by choosing size such that $(size + sizeof(*hdr)) > SIZE\_T\_MAX$ and $(size + sizeof(*hdr))\ \%\ SIZE\_T\_MAX < sizeof(*hdr)$. By allocating a heap object smaller than the *_mhead* metadata struct, the returned *dat* pointer allows the attacker to execute a heap overflow. If another object O_2 is allocated adjacent to the "too-small" object O_1, then writes to O_1 will overwrite the metadata of O_2, allowing the attacker to implement the heap exploit from project 1.

Problem 4

a. Sound: possible. Complete: impossible.

   Explanation: false alarms can occur in programs with errors or without errors, so soundness guarantees nothing about false alarms. Completeness by definition prevents false alarms.

b. Soundness is more important. It catches all errors along with some “false positives” which will waste people’s time or block harmless apps. Completeness would allow “false negatives,” undetected errors, which could allow insecure apps to be downloaded. That’s a much bigger problem in this case; companies don’t like getting hacked.

c. If the loop can be proved to never terminate (if, for example, x gets reset to 0 every iteration), then this program is secure. If this loop can be proved to terminate, then execution will continue to the insecure code, so the program is insecure. As a bonus, we’ve solved the halting problem!

Problem 5

A. If *d.idx* had value above the threshold, the function would return after line 1. Therefore at line 2, an upper bound has been found for *d.idx*, so its value is *upper_check*. It doesn't check for a lower bound, so it's not clean.

B. Since the abstract value of *d.idx* is *upper_check* but not *clean*, we've never lower-bounded *d.idx*, introducing a vulnerability in line 2. This line should produce an "array index possibly negative" error, since negative array indices allow reading arbitrary memorybelow the array.

C. This analysis will produce a "potential memcpy overflow" error from the third parameter to memcpy. 'skb' comes off the card tainted, so all its subfields are also tainted. Assuming no comparisons are run in the interim, the integer 'len' field is tainted when we arrive at the memcpy line. Memcpy is a trusting sink, so the analysis throws this error upon using $len - 1$ as the memcpy size. If $len - 1 > sizeof(cmd.parm.setup.phone)$ then this code introduces a buffer overflow.

Problem 6

a. On fork, all three uids are inherited from the parent process – so, n.

b. a. For a non-root process, the *setuid* syscall can change the euid to the ruid or suid (m in both cases), or not change it at all, leaving it with value n.

   b. A root process can set the euid to be any possible, valid uid it likes (including 0)

c. Assigning distinct, non-root uids allows the OS to restrict program B's access to program A's files using Unix access control, for example by allowing only $A.uid$ to write program A's config files, and only $A.uid$ to read program A's secrets. Two programs running under the same uid cannot be distinguished from each other by the file access control system.

d. $uid = 0$ is reserved for the root user. Running any process as root allows it both read and write access to every file, which prevents isolating it from any other program's files.

e. After a fork, the new process inherits its uid from its parent, so in this case the new process will be running as root. As explained in part (d), running arbitrary programs as root is Bad Times, so the zygote uses setuid to set the process's euid to non-root. The euid is what determines root priveleges, so this works.

f. a. On. The user, who is probably not root, needs to change her password, which requires writing to the password file. Only root can write to this file, so the setuid bit allows the user to run passwd as root (the owner of the passwd program).

   b. Since passwd is root-owned and has $setuid = 1$, but any user can execute it (since all users have passwords), exploits in the source code are particularly dangerous. If the user can trick the program into executing their own code through (e.g.) a buffer overflow, that code will run as root.