**Question 1 - multiple choice, shuffle**

Three of the following are classic security properties; which one is not?

\*A: Correctness

B: Confidentiality

C: Integrity

D: Availability

**Question 2 - multiple choice, shuffle**

What was the first buffer overflow attack?

A: SQL Slammer

B: Code Red

\*C: Morris Worm

D: Love Bug

**Question 3 - multiple choice, shuffle**

The stack is memory for storing

\*A: Local variables

B: Global variables

C: Program code

D: Dynamically linked libraries

**Question 4 - multiple choice, shuffle**

Why is it that the compiler does not know the absolute address of a local variable?

\*A: As a stack-allocated variable, it could have different addresses depending on when its containing function is called

B: Programs are not allowed to reference memory using absolute addresses

Feedback: Memory may be referenced by (absolute) address, at run-time, but those addresses may need to be computed based on run-time information.

C: The size of the address depends on the architecture the program will run on

Feedback: this is a true statement, but not salient to the question

D: Compiler writers are not very good at that sort of thing

**Question 5 - multiple choice, shuffle**

When does a buffer overflow occur, generally speaking?

\*A: when a pointer is used to access memory not allocated to it

B: when writing to a pointer that has been freed

Feedback: this is a "use after free", not a buffer overflow

C: when copying a buffer from the stack to the heap

Feedback: a copy might overflow the target buffer, but copying in general is not a problem.

D: when the program notices a buffer has filled up, and so starts to reject requests

Feedback: languages with bounds checking, and thus able to notice when a buffer "fills up", would prevent the actual overflow

**Question 6 - multiple choice, shuffle**

How does a buffer overflow on the stack facilitate running attacker-injected code?

\*A: By overwriting the return address to point to the location of that code

B: By changing the name of the running executable, stored on the stack

Feedback: Changing the name would not change the code itself (or what it does)

C: By writing directly to the instruction pointer register the address of the code

Feedback: Attacks cannot directly write to registers, only memory (that could be subsequently loaded into registers by the existing program)

D: By writing directly to %eax the address of the code

Feedback: Attackers cannot write to registers directly, only memory (that could be subsequently loaded into registers by the existing program)

**Question 7 - multiple choice, shuffle**

What is a nop sled?

A: It is another name for a branch instruction at the end of sequence of nops

B: It is an anonymous version of a mop sled

\*C: It is a sequence of nops preceding injected shellcode, useful when the return address is unknown

Feedback: The sequence nop instructs "sleds" the instruction pointer to the actual attacker code of interest

D: It is a method of removing zero bytes from shellcode

Feedback: A nop sled is not a method, but a sequence of instructions placed by the attacker

**Question 8 - multiple choice, shuffle**

The following program is vulnerable to a buffer overflow (assuming the absence of automated defenses like ASLR, DEP, etc., which we introduce in the next unit). What is the name of the buffer that can be overflowed?

#include <stdio.h> #include <string.h> #define S 100 #define N 1000 int main(int argc, char \*argv[]) { char out[S]; char buf[N]; char msg[] = "Welcome to the argument echoing program\n"; int len = 0; buf[0] = '\0'; printf(msg); while (argc) { sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1]); argc--; strncat(buf,out,sizeof(buf)-len-1); len = strlen(buf); } printf("%s",buf); return 0; }

\*A: out

B: buf

Feedback: The call to strncat respects the bounds of buf

C: msg

Feedback: msg never is written to

D: len

Feedback: len is not used as a buffer

**Question 9 - multiple choice, shuffle**

Here is the same program as the previous question. What line of code can overflow the vulnerable buffer?

#include <stdio.h> #include <string.h> #define S 100 #define N 1000 int main(int argc, char \*argv[]) { char out[S]; char buf[N]; char msg[] = "Welcome to the argument echoing program\n"; int len = 0; buf[0] = '\0'; printf(msg); while (argc) { sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1]); argc--; strncat(buf,out,sizeof(buf)-len-1); len = strlen(buf); } printf("%s",buf); return 0; }

A: printf(msg)

Feedback: msg never changes so there is no attacker-controlled overflow

\*B: sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1]);

Feedback: This can overrun out, which is of limited size, e.g., by having a very large command-line argument

C: strncat(buf,out,sizeof(buf)-len-1);

Feedback: This stays in the bounds of buf

D: printf("%s",buf);

Feedback: buf was correctly constructed earlier, so printf behaves correctly

E: len = strlen(buf);

Feedback: buf was correctly constructed earlier, so strlen behaves correctly.

**Question 10 - multiple choice, shuffle**

Recall the vulnerable overflow from the previous two questions. We can change one line of code and make the buffer overrun go away. Which of the following one-line changes, on its own, will eliminate the vulnerability?

#include <stdio.h> #include <string.h> #define S 100 #define N 1000 int main(int argc,char \*argv[]) { char out[S]; char buf[N]; char msg[] = "Welcome to the argument echoing program\n"; int len = 0; buf[0] = '\0'; printf(msg); while (argc) { sprintf(out,"argument %d is %s\n",argc-1,argv[argc-1]); argc--; strncat(buf,out,sizeof(buf)-len-1); len = strlen(buf); } printf("%s",buf); return 0; }

\*A: change sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1])

to snprintf(out, S, "argument %d is %s\n", argc-1, argv[argc-1])

Feedback: This removes the overflow directly - snprintf limits the writes to the buffer to be no more than its length (N).

B: change printf("%s",buf) to printf(buf);

Feedback: This doesn't help; in fact, it introduces another possible bug.

C: change printf(msg) to printf("%s",msg);

Feedback: This has no real effect; msg never changes.

D: change char msg[] = "Welcome to the argument echoing program\n" to char msg[42] = "Welcome to the argument echoing program\n"

Feedback: This has no real effect; when you don't specify the length for a statically initialized string the compiler does it for you.

**Question 11 - multiple choice, no shuffle**

Recall the vulnerable program from the previous few questions. Which of the following attacks do you think the program is susceptible to?

#include <stdio.h> #include <string.h> #define S 100 #define N 1000 int main(int argc, char \*argv[]) { char out[S]; char buf[N]; char msg[] = "Welcome to the argument echoing program\n"; int len = 0; buf[0] = '\0'; printf(msg); while (argc) { sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1]); argc--; strncat(buf,out,sizeof(buf)-len-1); len = strlen(buf); } printf("%s",buf); return 0; }

A: code injection

Feedback: This is possible by overwriting the return address to point to injected code.

B: data corruption

Feedback: This is possible by overwriting data on overflow

C: reading arbitrary addresses in memory

Feedback: This is possible by injecting code that will read those addresses

\*D: all of the above

Feedback: All of the attacks are indeed possible! Code injection is possible by overwriting the return address to point to injected code; data corruption is possible by overwriting data on overflow; reading memory is possible by injecting code that will read the desired addresses

**Question 12 - multiple choice, no shuffle**

Recall the program again.

#include <stdio.h> #include <string.h> #define S 100 #define N 1000 int main(int argc, char \*argv[]) { char out[S]; char buf[N]; char msg[] = "Welcome to the argument echoing program\n"; int len = 0; buf[0] = '\0'; printf(msg); while (argc) { sprintf(out, "argument %d is %s\n", argc-1, argv[argc-1]); argc--; strncat(buf,out,sizeof(buf)-len-1); len = strlen(buf); } printf("%s",buf); return 0; }

If we changed $$\color{red}{\verb|printf("%s",buf)|}$$ to $$\color{red}{\verb|printf(buf)|}$$ then the program would be vulnerable to what sort of attack?

A: heap overflow

\*B: format string attack

C: use-after-free attack

D: all of the above

**Question 13 - multiple choice, shuffle**

Exploitation of the Heartbleed bug permits

A: a kind of code injection

B: a format string attack

\*C: a read outside bounds of a buffer

D: overwriting cryptographic keys in memory

**Question 14 - multiple choice, shuffle**

Why is it that anti-virus scanners would not have found an exploitation of Heartbleed?

\*A: Anti-virus scanners tend to look for viruses and other malicious

code, but Heartbleed exploits steal secrets without injecting any code

Feedback: Note that an IDS might look for a suspicious exploit pattern, but anti-virus and IDS, while related, are not the same thing

B: Heartbleed exploits are easily mutated so the files they leave

behind do not appear unusual

Feedback: Heartbleed exploits do not leave files behind

C: It's a vacuous question: Heartbleed only reads outside a buffer, so

there is no possible exploit

Feedback: reading outside a buffer cannot be used to inject code, but it can be used to steal secrets, so this statement is false

D: Heartbleed attacks the anti-virus scanner itself

Feedback: Heartbleed attacks servers running the buggy version of SSL

**Question 15 - multiple choice, shuffle**

An integer overflow occurs when

\*A: an integer expression's result "wraps around"; instead of creating a very large number, a very small (or negative) number ends up getting created

B: an integer is used to access a buffer outside of the buffer's bounds

Feedback: overflowed integers may be used to access buffers out of bounds, but that is secondary to the fact that the integer itself is what is overflowed

C: an integer is used as if it was a pointer

Feedback: this could happen after an integer overflow, but it does not define what an integer overflow is

D: there is no more space to hold integers in the program