**Question 1 - multiple choice, shuffle**

A program indexes a buffer after a pointer to that buffer has been used as a parameter to the $$\color{red}{\verb|free()|}$$ function. This is

\*A: A violation of temporal memory safety

Feedback: Use of a buffer beyond its lifetime is a temporal safety issue

B: Correct behavior

Feedback: This is a violation of temporal safety

C: A violation of spatial memory safety

Feedback: A violation of spatial safety would exceed the bounds of the buffer

D: An information flow violation

Feedback: Information flow violations require sensitive information to flow to an unauthorized channel

**Question 2 - checkbox, shuffle, partial credit**

When could an integer overflow impact memory safety?

\*A: If the integer was used to perform pointer arithmetic

Feedback: if we did something like p = p+i where i is an overflowed integer then we could access outside of p's expected bounds

\*B: If the integer is passed as an argument to $$\color{red}{\verb|malloc()|}$$

Feedback: then the integer value passed to malloc could differ from the integer used to iterate over the buffer (e.g., it could have been multiplied by a data size)

C: If the integer was passed as a parameter to $$\color{red}{\verb|printf()|}$$

Feedback: printf does not use its integer parameters to access memory

D: If the integer is used as the denominator in a division expression

Feedback: This will cause a divide-by-zero error, but will not directly impact memory safety

E: If the integer was passed as a parameter to $$\color{red}{\verb|open()|}$$

Feedback: open does not use its integer parameters to access memory

**Question 2 - checkbox, variation 1, shuffle, partial credit**

When could an integer overflow impact memory safety?

\*A: If the integer was used to perform pointer arithmetic

Feedback: if we did something like p = p+i where i is an overflowed integer then we could access outside of p's expected bounds

\*B: If the integer is passed as an argument to $$\color{red}{\verb|strncat|}$$

Feedback: If an attacker can overflow n, e.g., making it negative, in a call like strncat(d,s,len-n), then we can overflow d.

C: If the integer is used as the denominator in a division expression

Feedback: This will cause a divide-by-zero error, but will not directly impact memory safety

D: If the integer was passed as a parameter to $$\color{red}{\verb|open()|}$$

Feedback: open does not use its integer parameters to access memory

E: Integer overflows never impact memory safety

Feedback: Integer overflows can be by design in some algorithms, but can impact memory safety when the integer is used in a way that interacts with memory

**Question 2 - checkbox, variation 2, shuffle, partial credit**

When could an integer overflow impact memory safety?

\*A: If the integer was used to index into an array

Feedback: then the integer value may not be correct when indexing into memory, e.g., if it was unsigned, and the overflow caused it to be negative

\*B: If the integer is passed as an argument to $$\color{red}{\verb|malloc()|}$$

Feedback: then the integer value passed to malloc could differ from the integer used to iterate over the buffer (e.g., it could have been multiplied by a data size)

C: Integer overflows never impact memory safety

Feedback: Integer overflows can be by design in some algorithms, but can impact memory safety when the integer is used in a way that interacts with memory

D: If the integer was passed as a parameter to $$\color{red}{\verb|printf()|}$$

Feedback: printf does not use its integer parameters to access memory

E: If the integer is used as the denominator in a division expression

Feedback: This will cause a divide-by-zero error, but will not directly impact memory safety

**Question 3 - checkbox, shuffle, partial credit**

Which of the following are true about a language that uses garbage collection or some other automatic means (e.g., reference counting) for memory management? (Select all that apply.)

\*A: The language will not have temporal memory safety violations

Feedback: The garbage collector will ensure that memory is only deallocated when it is not reachable, and this decision is not left up to the programmer

B: The language will not have spatial memory safety violations

Feedback: Garbage collection does not enforce accesses are in bounds

C: The language will use primarily static types (i.e., types that are checked primarily before the program runs)

Feedback: Static typing is unrelated to GC and reference counting; e.g., Python is dynamically typed and uses automatic memory management

**Question 3 - checkbox, variation 1, shuffle, partial credit**

Which of the following are true about a language that uses garbage collection or some other automatic means (e.g., reference counting) for memory management? (Select all that apply.)

\*A: The use of automatic memory management will provide a safety benefit, but typically at the cost of some performance

Feedback: This is because the garbage collectors usually perform periodic work (e.g., GC tracing or reference counting) to find non-dead memory

B: The language will not have type safety violations

Feedback: Garbage collection does not enforce type safety on its own, though it can prevent some type safety violations (which occur via temporal safety violations)

C: The language will not have spatial memory safety violations

Feedback: Garbage collection does not enforce accesses are in bounds

**Question 4 - multiple choice, shuffle**

Consider the following code:

 char \*foo(char \*buf) { char \*x = buf+strlen(buf); char \*y = buf; while (y != x) { if (\*y == 'a')break; y++; } return y; } void bar() { char input[10] = "leonard"; foo(input); }

The definition of spatial safety models pointers as capabilities, which are triples *(p,b,e)* where *p* is the pointer, *b* is the base of the memory region the pointer is allowed to access, and *e* is the extent of that region. Assuming characters are 1 byte in size, what is a triple *(p,b,e)* for the variable $$\color{red}{\verb|y|}$$ when it is returned at the end of the code?

\*A: (&input+4,&input,&input+10)

Feedback: y starts out as pointing to the input[] buffer, which has space for 10 characters. y is incremented 4 times, until it reaches the 'a' in the string.

B: (&input+4,&input,&input+7)

Feedback: While the length of the string is 7 characters, the full extent of the buffer is 10 characters, per the declaration of input[]

C: (y,&input,buf)

Feedback: The extent component is the end of the buffer, but buf points to the start of it

D: (&input+4,0,sizeof(input))

Feedback: The base and extent are defined as pointers, not indexes into the buffer

**Question 5 - checkbox, shuffle, partial credit**

Which of the following are true about a type-safe language? (Select all that apply.)

\*A: The language is also memory safe

Feedback: Type safety is stronger than memory safety

B: The language is object-oriented.

Feedback: Type safety and object orientation are independent qualities of a language.

C: The language is sometimes memory safe, but not always

Feedback: Type safe languages are always memory safe

D: The language is always *much* slower than a non-type safe language

Feedback: Some type-safe languages are much slower, but not all. Type-safe languages can be optimized to run within a couple of factors of C and/or C++, and even better when applied to program domains for which they were designed

**Question 5 - checkbox, variation 1, shuffle, partial credit**

Which of the following are true about a type-safe language? (Select all that apply.)

A: The language is object-oriented.

Feedback: Type safety and object orientation are independent qualities of a language.

\*B: The language may be used to enforce information flow security, depending on the type system

Feedback: This is done in the JIF programming language

C: The language is sometimes memory safe, but not always

Feedback: Type safe languages are always memory safe

D: The language's types are static, i.e., checked by the compiler before running the program

Feedback: Type safety applies to dynamically typed languages as well (whose types are checked during execution)

**Question 5 - checkbox, variation 2, shuffle, partial credit**

Which of the following are true about a type-safe language? (Select all that apply.)

A: The language is object-oriented.

Feedback: Type safety and object orientation are independent qualities of a language.

\*B: The language may be used to enforce information flow security, depending on the type system

Feedback: This is done in the JIF programming language

C: The language is always *much* slower than a non-type safe language

Feedback: Some type-safe languages are much slower, but not all. Type-safe languages can be optimized to run within a couple of factors of C and/or C++, and even better when applied to program domains for which they were designed

D: The language is sometimes memory safe, but not always

Feedback: Type safe languages are always memory safe

**Question 6 - multiple choice, shuffle**

An engineer proposes that in addition to making the stack non-executable, your system should also make the heap non-executable. Doing so would

\*A: Make the program more secure by disallowing another location for an attacker to place executable code

Feedback: Then attacker data in the heap cannot be executed, enforcing (W xor X) / DEP for the entire program

B: Not make the program more secure, because attacker-controlled data cannot be stored in the heap

Feedback: Attacker controlled data can be stored in the heap

C: Ensure that only the correct amount of data was written to a heap-allocated block, preventing heap overflows

Feedback: Non-executable memory must still be bounds checked

D: Ensure that memory is always deallocated

Feedback: Non-executable memory must still be deallocated properly

**Question 7 - multiple choice, shuffle**

What is the *best* choice of value for a stack canary, of the following options?

\*A: A random value

Feedback: The canary should be unpredictable, so the attacker cannot easily guess it if he must overwrite it during an attack

B: A predictable value

Feedback: If the canary is predictable then the attacker can provide it

C: The constant 0

Feedback: If the canary is a constant then the attacker can provide it. Note that 0 is slightly better than other constants because it would complicate copying in attacker data via strcpy

D: The constant 7

Feedback: If the canary is a constant then the attacker can provide it

**Question 8 - multiple choice, shuffle**

A return-to-libc attack does not require that the attacker inject executable code into the vulnerable program. Which of the following is the *most important* reason that return-to-libc attacks are useful to the attacker?

\*A: There is no need to be able to execute (writable) data

Feedback: The attacker does not need to inject executable code into an writable buffer, therefore they can exploit systems that enforce (W xor X) / DEP

B: The injected code might have bugs

Feedback: Attackers can solve this problem by fixing the code they inject; it is independent of what defenses might be enabled at the target

C: The code in libc is better than code the attacker would write

Feedback: Attackers can solve this problem by writing better code; the concern is independent of what defenses might be enabled at the target

D: There is no need to modify the application's executable code

Feedback: The attacker can compromise the program without modifying the application's executable code; code injection attacks, for example, do not modify the existing code

**Question 9 - multiple choice, shuffle**

In a return-oriented program (ROP), what is the role of the stack pointer?

\*A: It's like the program counter in a normal program

Feedback: the stack pointer is used to select the next instruction to execute via a 'ret'

B: It's like the frame pointer in a normal program

Feedback: The frame pointer demarcates the storage space for local variables

C: It's like the allocation pointer used by malloc()

Feedback: allocation pointers returned via malloc are for data storage

D: It's really no different than in a normal program

Feedback: the stack pointer is used for a very different purpose in ROP

**Question 10 - multiple choice, shuffle**

When enforcing Control Flow Integrity (CFI), there is no need to check that direct calls adhere to the control flow graph because:

\*A: CFI should be deployed on systems that ensure the code is immutable

Feedback: If the code cannot be changed then direct calls cannot be re-written to point to an attacker-supplied value

B: The attacker is not interested in corrupting direct calls

Feedback: The attacker would be happy to hijack control flow however he can achieve it

C: CFI should be deployed on systems that ensure the data is non-executable

Feedback: While true, this is not why CFI can ignore direct calls.

D: Programs that use CFI don't have direct calls

Feedback: Most programs have direct calls, and CFI ought to (and does) apply to most (or all) programs

**Question 11 - checkbox, shuffle, partial credit**

Recall that classic enforcement of CFI requires adding labels prior to branch targets, and adding code prior to the branch that checks the label to see if it's the one that is expected. Now consider the following program:

 int cmp1(char \*a, char \*b) { return strcmp(a,b); } int cmp2(char \*a, char \*b) { return strcmp(b,a); } typedef int (\*cmpp)(char\*,char\*); int bar(char \*buf) { cmpp p; char tmpbuff[512] = { 0 }; int l; if(buf[0] == 'a') { p = cmp1; } else { p = cmp2; } printf("%p\n", p); strcpy(tmpbuff, buf); for(l = 0; l < sizeof(tmpbuff); l++) { if(tmpbuff[l] == 0) { break; } else { if(tmpbuff[l] > 97) { tmpbuff[l] -= 32; } } } return p(tmpbuff,buf); }

To ensure that the instrumented program runs correctly when not being attacked, which of the following functions would have to be given the same label? Choose at least two, but no more functions than necessary.

\*A: cmp1

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

\*B: cmp2

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

C: strcmp

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

D: cmpp

Feedback: cmpp is not a function; it is a the type of a certain kind of function pointer

E: bar

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

**Question 11 - checkbox, variation 1, shuffle, partial credit**

Recall that classic enforcement of CFI requires adding labels prior to branch targets, and adding code prior to the branch that checks the label to see if it's the one that is expected. Now consider the following program:

 int cmp1(char \*a, char \*b) { return strcmp(a,b); } int cmp2(char \*a, char \*b) { return strcmp(b,a); } typedef int (\*cmpp)(char\*,char\*); int bar(char \*buf) { cmpp p; char tmpbuff[512] = { 0 }; int l; if(buf[0] == 'a') { p = cmp1; } else { p = cmp2; } printf("%p\n", p); strcpy(tmpbuff, buf); for(l = 0; l < sizeof(tmpbuff); l++) { if(tmpbuff[l] == 0) { break; } else { if(tmpbuff[l] > 97) { tmpbuff[l] -= 32; } } } return p(tmpbuff,buf); }

To ensure that the instrumented program runs correctly when not being attacked, which of the following functions would have to be given the same label? Choose at least two, but no more functions than necessary.

\*A: cmp1

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

\*B: cmp2

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

C: printf

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

D: strcmp

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

E: cmpp

Feedback: cmpp is not a function; it is a the type of a certain kind of function pointer

**Question 11 - checkbox, variation 2, shuffle, partial credit**

Recall that classic enforcement of CFI requires adding labels prior to branch targets, and adding code prior to the branch that checks the label to see if it's the one that is expected. Now consider the following program:

 int cmp1(char \*a, char \*b) { return strcmp(a,b); } int cmp2(char \*a, char \*b) { return strcmp(b,a); } typedef int (\*cmpp)(char\*,char\*); int bar(char \*buf) { cmpp p; char tmpbuff[512] = { 0 }; int l; if(buf[0] == 'a') { p = cmp1; } else { p = cmp2; } printf("%p\n", p); strcpy(tmpbuff, buf); for(l = 0; l < sizeof(tmpbuff); l++) { if(tmpbuff[l] == 0) { break; } else { if(tmpbuff[l] > 97) { tmpbuff[l] -= 32; } } } return p(tmpbuff,buf); }

To ensure that the instrumented program runs correctly when not being attacked, which of the following functions would have to be given the same label? Choose at least two, but no more functions than necessary.

\*A: cmp1

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

\*B: cmp2

Feedback: could be assigned to p, a function pointer and therefore an indirect branch target

C: strcmp

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

D: strcpy

Feedback: cannot be assigned to p, a function pointer and therefore an indirect branch target

E: cmpp

Feedback: cmpp is not a function; it is a the type of a certain kind of function pointer

**Question 12 - multiple choice, shuffle**

In your review of a program, you discover the following function:

 void aFunction(char \*buf) { static char BANNED\_CHARACTERS[] = { '>', '<', '!', '\*' }; int l = strlen(buf); int i; for(i = 0; i < l; i++) { int j; int k = sizeof(BANNED\_CHARACTERS) / sizeof(char); for(j = 0; j < k; j++) { if(buf[i] == BANNED\_CHARACTERS[j]) buf[i] = ' '; } } }

 How would you best describe what this function is doing?

\*A: Input sanitization by blacklisting

Feedback: if a potentially dangerous ("black") character, given in the list, is present then it is removed

B: Input validation by whitelisting

Feedback: whitelisting allows only values from an specified set

C: Spatial safety enforcement

Feedback: This does not enforce the accesses into a buffer are in bounds

D: Using a safe string library

Feedback: This code uses standard string library functions (like strlen) and does not implement safe strings

**Question 13 - multiple choice, shuffle**

A safe string library typically attempts to ensure which of the following?

\*A: That there is sufficient space in a source and/or target string to perform operations like concatenation, copying, etc.

Feedback: safe string libraries enforce spatial memory safety

B: That the strings have been properly sanitized

Feedback: String libraries are independent of the domain of strings they will be used in, and sanitization is specific to a particular context

C: That strings from the safe library can be freely passed to the standard string library functions, and vice versa

Feedback: Safe string libraries typically operate on strings with different data types from C strings, to be able to ensure safety

D: That wide (i.e., multibyte) character strings can be used where single-byte character strings are expected.

Feedback: Wide strings cannot always be used where single byte character strings are expected, and in any case this is not the concern of safe string libraries

**Question 14 - multiple choice, shuffle**

A project manager proposes a C coding standard where pointer variables must be assigned to $$\color{red}{\verb|NULL|}$$ after being passed to $$\color{red}{\verb|free()|}$$. Doing so:

\*A: Stops writes to stale pointer values that might otherwise succeed and result in program compromise

Feedback: Writing NULL means that a dereference will result in a crash, which is undesirable but nevertheless helps prevent exploitable vulnerabilities

B: Prevents memory leaks, thus avoiding potential denial of service

Feedback: In a garbage collected language, assigning NULL could prevent a leak, but not in C

C: Helps code readability, but not security

Feedback: This has low impact on code readability

D: Is a poor security decision, because NULL pointer dereferences could cause the program to crash

Feedback: Crashes are better than compromise

**Question 15 - multiple choice, shuffle**

A colleague proposes using a heap allocator that randomizes the addresses of allocated objects. This:

\*A: Will make the program more secure, because attackers frequently rely on predicting the locations of heap-allocated objects in exploits

B: Will make the program less secure, because the application will not be able to predict the locations of heap-allocated objects

Feedback: The application should not need to predict locations of its own data; to the extent that it matters, it can know/compute them

C: Will increase performance by keeping the cache sparsely populated

Feedback: Sparsely populating the cache is bad for performance, not good for it. Indeed, randomization might hurt spatial locality if objects allocated close in time would normally be placed near each other in the heap

D: Will have no impact on security or performance

Feedback: Randomization will improve security by making it more difficult for attackers to predict the location of heap allocated objects