Computer Science 5271
Fall 2024
Midterm exam (solutions)
October 23rd, 2024
Time Limit: 75 minutes, 1:00pm-2:15pm

- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to. Don't put any of your answers on this page.
- This exam contains 14 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- By signing below you certify that you agree to follow the rules of the exam, and that the answers on this exam are your own work only.

The exam will end promptly at 2:15pm. Good luck!

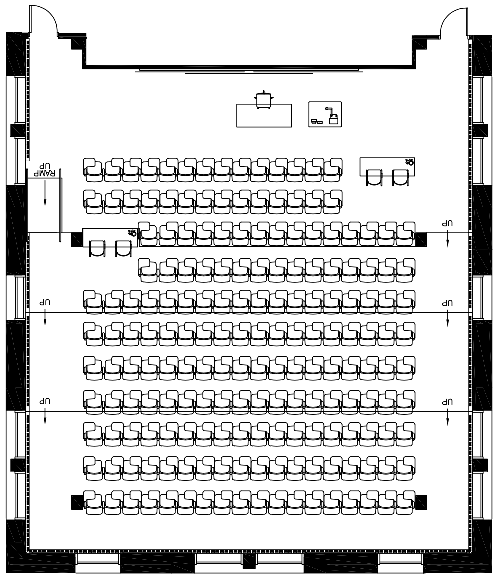
Your name (print): ______________________________

Your UMN email/X.500: ______________________________`@umn.edu`

Sign and date: ______________________________

Mark the icon corresponding to your seat:

Question	Points	Score
1	30	
2	20	
3	30	
4	20	
Total:	100	



1. (30 points) Multiple choice. Each question has only one correct answer: circle its letter.

   (a) The number 20 in decimal is represented as `0x14` in hex. All of the following pairs of 32-bit integers, shown in hex, would produce a value of `0x14` when multiplied using the rules of C for `unsigned int` on x86-64, **except**:

   A. `0xfffffffb` · `0xfffffffc`
   B. `0x80000002` · `0x8000000a`
   C. `0x33333333` · `0x00000003`
   D. `0x00000005` · `0x00000004`
   E. `0x80000005` · `0x00000004`

   The shortcut here is to look at the least significant parts of the product. Just like you may recall from decimal long multiplication in grade school, the least significant digit of a product depends only on the least significant digits of the factors. When applied to bits, this corresponds to the fact that the product is even if either of the factors is even, and odd only if they are both odd. (The odd hex digits are a superset of the odd decimal digits, namely `1`, `3`, `5`, `7`, `9`, `b` *(11),* `d` *(13), and* `f` *(15)). If the product is going to be* `0x14`*, one of the factors has to be even, which is true of those ending in* `c`, `2`, `a`*, and* `4`*, but crucially not of those ending in* `3`*. Since the second factor has only one non-zero digit, and there is a clear pattern, you may also have been able to compute that the third product will be* `0x99999999`*, not* `0x14`*. With a bit more computation, you can also see that the other products will all have a least-significant hex digit of* `4`*.* `2` *times* `a` *and* `5` *times* `4` *both multiply to 20.* `b` *times* `c` *is a somewhat harder calculation of* $11 \cdot 12 = 132 = 128+4 = 8 \cdot 16+4$*. An alternative way to figure out the product in A is to notice that the numbers would be interpreted as -4 and -5 respectively in twos-complement. The instructions specify unsigned multiplication, but twos-complement signed multiplication and unsigned multiplication are the same operation if you only keep the same number of output bits as you had input bits, so the low 32 bits of this product will be 20 as well.*

   (b) Suppose a local variable whose type is an array of 100 characters contains sensitive information in the form of short (3-6 character) printable strings separated by null bytes, and assume the platform is Linux/x86-64. If the function containing the variable also has a call to `printf` that is vulnerable to a format string attack, which of these format specifiers would be the best choice for an attacker to use repeatedly to dump the entire contents of the array?

   A. `%lx` B. `%c` C. `%ho` D. `%s` E. `%x`

   Repeated format specifiers are the normal way to leak information with a format string attack, but the key distinction here is how the format specifiers interpret the stack contents, since the goal is not to miss out on any of the strings in the array. The arguments to format specifiers, like other arguments passed on the stack, are laid out in units of the word size, 8 bytes on x86-64. If the format specifier ignores some of those bytes when printing, the attacker will miss out on some data. `%c` *and* `%s` *might both sound initially appealing because they relate to character data, but neither one works well.* `%c` *prints only the least significant byte of each 8 as a character, so 7/8 of the information is skipped. Because* `%s` *is for printing null-terminated strings, each* `%s` *could only print one string. But there is actually a more basic problem with* `%s`*: the argument to* `%s` *is a pointer to a null-terminated string, not the string contents itself. So a* `%s` *format specifier would try to interpret a mixture of string and null terminator bytes as a pointer, and probably just* *crash. Using an integer-oriented format specifier is a better strategy, since any byte values can be interpreted as integers; even if the output doesn't superficially look like strings, an attacker could use a simple script to convert the data into another base, split back into bytes, and recover the strings. So the attacker needs an integer format specifier that uses all 8 bytes of each stack argument slot.* `%ho` *only uses 2 bytes, since the* `h` *modifier treats the argument as a* `short`*.* `%x` *uses 4 bytes, since the unmodified specifiers correspond to a type* `int`*.* `%lx` *is best because it prints each 8-byte section of stack as an 8-byte* `long` *(and as a side benefit hex also makes the conversion to characters easier than other bases).*

(c) The set of all subsets of the letters A though J forms a lattice when the ordering operation $\sqsubseteq$ is defined to be the subset operation $\subseteq$. Under this definition, what is the least upper bound $\{A, B\} \sqcup \{E, F\}$?

A. $\varnothing$ (the empty set)

B. $\{A, B\}$

C. $\{A, B, E, F\}$

D. $\{A, B, C, D, E, F\}$

E. $\{A, B, C, D, E, F, G, H, I, J\}$

This kind of structure is called a subset lattice, and it is a very commonly occurring kind of finite lattice; specifically in security we've seen it occurs when you can have any combination of specialized compartments. It's not a coincidence that the common generic notation for lattice operators looks like squared-off versions of the standard set operations, since the least upper bound in a subset lattice always corresponds to set union $\cup$*. To be an upper bound, it needs to include all the elements that were in either of the inputs, but to be the least upper bound it shouldn't include any others.*

(d) Why would password capabilities be an appropriate kind of capability-based access control to be used in a large-scale networked system?

A. Object-aggregated authority management scales to having many objects

B. A centralized server can immediately revoke password capabilities

C. Capabilities automatically provide ambient authority

D. Password capabilities don't need to be managed by an OS kernel

E. Password capabilities cannot be transferred via network messages

Recall that password capabilities are contrasted with the more standard object capabilities. Both kinds of capabilities designate an object along with representing the permission to access that object. The difference is that object capabilities are managed on behalf of a process by an OS kernel to prevent forgery: the kernel has the authoritative information on which capabilities a process holds. The example of Unix file descriptors are like object capabilities in this way. By contrast, a password capability is just a string of bits that can be passed like any other data; a password capability is only hard to forge because the bit pattern is chosen unpredictably with high entropy, like a password (thus the name) or a cryptographic key. Answers A and C are the opposites of true statements about capabilities in general: capabilities are subject-aggregated rather than object-aggregated, and capabilities are desirable because they avoid depending on ambient authority. Answers B and E are the opposites of true statements about password capabilities: password capabilities aren't based on a centralized server and do not directly support revocation, and password capabilities can be transferred in network messages.

(e) Random stack canaries and ASLR share all of the following features **except**:

A. They are more resistant to guessing if they are re-randomized frequently

B. They make some control-flow hijacking attacks more difficult

C. They require kernel support to implement

D. They are more resistant to guessing if they have more entropy

E. Their protection is compromised if information leaks to an attacker

Answers B states the general purpose of both stack canaries and ASLR. Answers A, D, and E are all features that random canaries and ASLR have in common because they are based on randomization that needs to be unknown to an attacker to be an effective defense. By contrast C is not true of either defense. Stack canaries are usually implemented by a compiler with no OS involvement at all. ASLR could potentially be implemented either with user-space logic or with changes to a kernel; it has been common for ASLR to be implemented in the kernel because that is an easy place to enable it without having to change or recompile user-space programs.

(f) All of these attack techniques were predecessors of ROP, **except**:

A. Control-flow bending

B. Return to libc

C. ret2pop

D. Chained return to libc

E. Stack smashing

"Stack smashing" is a synonym for a stack buffer overflow that overwrites a return address, which is the most basic kind of attack of this kind. Return to libc, chained return to libc, and ret2pop are all simple kinds of code retuse attacks involving return instructions that predate the general form of ROP. On the other hand, control-flow bending is a more recent and advanced form of attack that uses only valid control-flow transfers.

(g) Suppose a program on an x86-32 platform has a hard-to-control memory safety vulnerability that leads to a return address being overwritten by a uniformly random 32-bit value. An attacker is able to set up a heap spray by allocating 1000 memory objects, each of which is 1 MiB ($2^{20} = 1048576$ bytes) long, containing a NOP sled and a 100 byte-long shellcode. These objects are placed at non-overlapping locations in the address space. If the attacker repeats the attack 10 times, what is the probability of succeeding at least once?

A. $(1 - (1000 \cdot (2^{20} - 100)/2^{32}))^{10} \approx 6\%$

B. $10 \cdot 100 \cdot 2^{20}/2^{32} \approx 24\%$

C. $(10 \cdot 100 \cdot 1000/2^{20})^{10} \approx 62\%$

D. $1 - (1 - ((1000 - 100) \cdot (2^{20})/2^{32}))^{10} \approx 92\%$

E. $1 - (1 - (1000 \cdot (2^{20} - 100 + 1)/2^{32}))^{10} \approx 94\%$

The best layout for each of the megabyte-long attack objects is to have the shellcode at the end and a NOP sled filling all the space before it: in this layout, jumping to any byte of the NOP sled or the first byte of the shellcode will be sure to execute the shellcode correctly, while jumping into the middle of the shellcode will usually not work. So we can estimate the number of usable jump targets as $2^{20} - 100 + 1$ *for each of the 1000 objects, which is about 1 billion. The total address space is* 2^{32} *or about 4 billion bytes, so jumping to a random byte location will give control to the attacker with probability about* $1/4$. *If the* *attacker gets to try this 10 times and only has to succeed once, those are pretty good odds. To calculate them precisely, think about the complementary probabilities: the attacker only fails if they fail on all 10 tries, and those tries are independent, so the failure probabilities of about 3/4 each just multiply, so we take the 10th power, and then the final success probability is 1 minus the total failure probability. These operations are answer E.*

(h) All of the following situations are specified to constitute undefined behavior in the C language standard **except**:

A. Dereferencing a null pointer

B. Accessing a memory region after it has been `free()`d

C. An unhandled case in a `switch` statement

D. Accessing outside the bounds of an array

E. Integer overflow of a signed integer

Undefined behaviors are ones where the language standard says nothing about what should happen. This makes them always risky from a security perspective, and so from a programming perspective they should be avoided. But the details of what will actually happen depends on the design and choices made by a compiler. Answers A, B, and D are both undefined and likely to cause the program to crash. Integer overflow in E would usually not be an operation that itself causes a crash (though it could on some less common CPUs), but the fact that it is an undefined behavior means that compilers are allowed to optimize under the assumption that it won't happen. By comparison, the language requires (defines) that values in a switch that do not correspond to any of the listed cases cause control to go to then end of the switch.

(i) Applying the metric of net risk reduction implies that a security protection becomes more worthwhile when any of these happen, **except**:

A. The expected damage caused by an attack increases

B. The attack becomes more frequent

C. The cost of carrying out the attack goes up

D. The defense becomes less expensive

Net risk reduction is a specific kind of cost benefit analysis, where we measure the benefit from a defense in reducing the expected damage from an attack (risk reduction) against the cost of the defense. Scenarios A and B would increase the expected damage, and so increase the benefit of a defense, while scenario D is that the cost decreases, so any of these make the cost-benefit tradeoff more favorable. Choice C talks about the cost of carrying out the attack, which is a cost that doesn't directly figure into a net risk reduction calculation, since net risk reduction is about the defender's choices rather than the attacker's choices. However if you think about an indirect effect, it would probably go in the opposite direction: if the cost of carrying out an attack goes up, attackers will attempt the attack less often, and so the risk and net risk reduction would go down.

(j) Some laptops and smartphones now encourage users to log in via facial recognition or a fingerprint instead of with a password or PIN. These are examples of:

A. Single sign-on

B. Two-factor authentication

C. Biometric authentication

D. Compromise recording

E. CAPTCHAs

Facial recognition and fingerprints are the most common examples of biometric authentication. If for instance you had to use both your fingerprint and a password, that would be an instance of two-factor authentication, but the question asked about a biometric factor replacing a knowledge-based factor (which is probably more common in consumer devices: it is a primary selling point of biometric authentication that it is more convenient than a password.)

2. (20 points) A race condition attack.

The following high-level C code attempts to copy the contents of one temporary file belonging to the Alice (username `alice`) into a new file that will also be owned by Alice. However, you may be able to see that it has TOCTTOU/race condition problems.

```
char data[32000]; size_t data_len;
void copy_alice_file(char *input_file, char *output_file) {
      /*** point A ***/
    if (!file_exists(input_file))
        print_error_and_exit();
      /*** point B ***/
    if (!is_alice_owned_and_readable(input_file))
        print_error_and_exit();
      /*** point C ****/
    if (!file_exists(output_file))
        create_alice_file(output_file);
      /*** point D ****/
    FILE *input_fh = fopen(input_file, "r");
    if (!input_fh)
        print_error_and_exit();
    read_data(input_fh, data, &data_len, sizeof(data));
    fclose(input_fh);
      /*** point E ***/
    FILE *output_fh = fopen(output_file, "w");
    if (!output_fh)
        print_error_and_exit();
    write_data(output_fh, data, data_len);
    fclose(output_fh);
}

int main(int argc, char **argv) {
    /* ... */
    copy_alice_file("/tmp/alice.in", "/tmp/alice.out");
    return 0;
}
```

Suppose that the program containing this code runs with superuser privileges, and your goal as an attacker with the username `bob` is to trick the program into doing something else. Specifically Bob wants the program to copy the contents of the secret file `/etc/shadow`, which contains information about other users' passwords, into a file that he (Bob) can read. (To start, `/etc/shadow` is only readable by the superuser.) Assume that Bob triggers the execution of this program at a time when initially neither the input file `/tmp/alice.in` nor the output file `/tmp/alice.out` exists yet. But a different file named `/tmp/alice-recipes` owned by Alice with 0600 permissions does exist. Bob is able to run other programs, using his write access to `/tmp`, at the same time this code is running. In particular, to achieve his attack, Bob will try to get certain file system operations to occur in between the vulnerable program's operations, namely at the points marked point A through point E.

In the parts below, describe which racing attacker actions Bob should take at each point for a successful attack. You may not need to use all of the points. Suggestion: use symbolic links.

Here is one possible complete answer:

(a) At point A:
Create `/tmp/alice.in` as a symlink to `/tmp/alice-recipes`.

(b) At point B:

(c) At point C:
Create `/tmp/alice.out` as a file owned by `bob` with `0600` permissions.

(d) At point D:
Redirect `/tmp/alice.in` to point to `/etc/shadow`.

(e) At point E:

More generally, we were looking for operations in three areas to make a successful attack:

1. *Pass the checks after points A and B by creating a file as* `/tmp/alice.in` *that would pass those checks. Since later steps of the attack require this file to be a symlink, the requirements can be fulfilled in a single step by creating the file as a symlink to* `/tmp/alice-recipes`*. Alternatively, Bob could create any file at all at point A, and then later replace it at step B. Creating a symlink to another file owned by Alice is the best approach because could not create a new file owned by Alice himself.*
2. *Update the* `/tmp/alice.in` *symlink to point to the file whose contents Bob wants to leak,* `/etc/shadow`*.* `/etc/shadow` *is not owned by Alice, so this check has to come after point B, but it has to come before the program opens the file for reading, so point C and point D are the two viable locations for this change.*
3. *Create an output file for the leaked data to be written into. Bob has to create some file with the name* `/tmp/alice.out` *no later than point C, because if the file does not exist after point C, the program will create an Alice-owned file at this location that Bob will not be able to read. The final Bob-readable file needs to be created before step E when the program will open it. Our solution shows one action creating a suitable file at step C, but creating a file earlier and then changing it later could also work.*

Assume that all of the functions whose names contain underscores do what their name sounds like they do: The function `file_exists` returns true if a file exists with a given pathname, and false otherwise. The function `print_error_and_exit` prints an error message and then causes the program to exit. The function `is_alice_owned_and_readable` returns true if its pathname argument is owned by `alice` and has read permissions for `alice`. Otherwise it returns false. The function `create_alice_file` creates a file with the given name suitable for storing information private to Alice: the owner of the file is `alice` and only `alice` has read or write permissions. The function `read_data` reads the contents from an open file handle into a memory buffer, keeping track of the amount of data it reads. You don't have to worry about the possibility of the file contents being bigger than the buffer. The function `write_data` is the matching operation to `read_data` and similar to the standard library function `fwrite`, writing data from memory back into a file handle open for writing.

The standard library function `fopen` opens a file handle used to read from or write to a file (specified by the second argument). It returns a null pointer if the file cannot be opened, such as if it does not exist. The standard library function `fclose` is used to close a file handle opened by `fopen`. We have left off error handling for `fclose` because it is not important to this vulnerability.

3. (30 points) Function preconditions.

   Each of the following short C functions performs some operations that are potentially unsafe, but could be performed correctly if appropriate properties of the function arguments, *preconditions*, are checked by the code calling the function. For each function, write one or more preconditions that are sufficient to guarantee that no safety or security problems will happen when running the function, but allow appropriate uses of the function to occur. Use the syntax of C to represent the preconditions whenever possible, and in other cases write clear text such as might appear in documentation.

   You don't have to mention any properties that are already checked with the `assert` function inside the function, and the number of blank lines is not intended to signal the number of preconditions we expect. We've done the first function as an example.

   `n >= 0`

   ```
   unsigned int fib(int n) {
       if (n == 0 || n == 1) {
           return 1;
       } else {
           return fib(n - 1) + fib(n - 2);
       }
   }
   ```

   (a) `strlen(s1) + strlen(s2) + strlen(s3) <` (size of the buffer pointed to by `buf`)

   ```
   void strcat3(char *buf, const char *s1, const char *s2, const char *s3) {
       assert(buf && s1 && s2 && s3);
       strcpy(buf, s1);
       strcat(buf, s2);
       strcat(buf, s3);
   }
   ```

   The `assert` *here already checks that the pointers are non-null, so the most important precondition we were looking for was ensuring that there would be enough space in the output buffer for all the strings that are going to be copied into it.* `strlen` *counts the number of bytes in a string up to but not including the null terminator, so it is the right operation to use to compute how many characters will be copied from* `s1`*,* `s2`*, and* `s3`*. But it would not make sense to call* `strlen` *on* `buf`*, since that would count the length of the string that was already written there; by contrast we probably don't want to assume that the output buffer has been initialized at all, since the purpose of this code is to overwrite it. (Or it could have previously held a shorter string.) We gave full credit for* `sizeof(buf)`*, but note that* `sizeof` *would only do the right thing if the buffer was a local or global array whose size was fixed by the compiler, and* `sizeof` *was applied to that array. Applying* `sizeof` *to the character pointer* `char *buf` *would not give the right result, since pointers* *are always a fixed size (8 bytes on x86-64), regardless of what they're pointing to. After all the copying is done there also needs to be space for a null terminator byte after the copy of* `s3`*: you can handle this by using a strict* `<` *comparison, or by adding 1 to the sum of the lengths and using* `<=`*.*

(b) `size <= 0x1fffffffffffffff`

```
long *alloc_and_zero_array(size_t size) {
    long *p = malloc(size * sizeof(long));
    assert(p);
    for (size_t i = 0; i < size; i++) {
        p[i] = 0;
    }
    return p;
}
```

The main problem we should worry about here is integer overflow in the multiplication leading to buffer overflow when writing to a too-small array. Our sample answer shows a numeric bound that is right when `size_t` *is* `unsigned long` *and* `long`*s are 64 bits, as they are on x86-64. We did not intend to require that your answer be portable to other platforms, but many students chose to express this result in terms of the macros for the sizes of various types. For this style our preferred answer would be* `SIZE_MAX/sizeof(long)`*, since* `SIZE_MAX` *is the maximum value of a* `size_t` *variable. Some other pitfalls to mention: this wouldn't have been a good case for trying to detect overflow by doing calculations on a different type, since integers wider than 64 bits aren't consistently available, and* `double` *floating point variables can't represent all 64-bit integer values exactly. It is also not correct in general to base an overflow check on assuming that an overflowed result will have a different ordering relative to the operands as a non-overflow result: this is true about adding two integers, but not may other operations. For instance in this case, the computation* `8 * size` *could overflow and still be larger than* `size`*; for instance* `8 * 0x8800000000000000`*' gives* `0x4000000000000000`*. On current x86-64 systems such overflows would probably still be large enough that* `malloc` *would fail, but they would become an exploitable vulnerability if future systems support larger allocations.*

(c) `p != NULL`

`p` points to a writeable `char`

```
void poke(char *p, char c) {
    *p = c;
}
```

Checking for the pointer `p` *to be non-null was the most commonly-given part of this answer. But being non-null is not enough to ensure that a pointer can be safely written to: for instance it might point to an object that was allocated and then freed, or it could point to a read-only string constant.*

(d) `p != NULL`

`p` points to a readable `char`

```
char peek(char *p) {
    return *p;
}
```

This question was intended to be similar to the previous one, except that the memory access is a read rather than a write.

(e) `x != 0`

```
int abs_dynamic(int x) {
    int *pos_ptr = malloc(sizeof(int));
    int result;
    assert(pos_ptr);
    if (x >= 0) {
        *pos_ptr = x;
        result = *pos_ptr;
        free(pos_ptr);
    }
    if (x <= 0) {
        *pos_ptr = -x;
        result = *pos_ptr;
        free(pos_ptr);
    }
    return result;
}
```

This absolute value function was written to weirdly require dynamic memory allocation, and the problem we were hoping you would notice is that the non-negative and non-positive cases will both be executed when the input is 0, leading to both a use-after-free problem and a double-free problem. This would probably be classified as a bug, but it can be avoided by not passing 0 as an input.

It wasn't part of our intended answer, but some students also pointed out that the value of `0x80000000` *for* `x` *also has a problem. It isn't something that would likely cause a crash on most systems, but this most negative twos-complement value has no corresponding representable positive value, so the usual behavior of negation is to overflow back to leaving the value unchanged. This is undefined behavior under the rules of C, and it also risks triggering problems in code that uses this function which would otherwise assume that the result of this function is non-negative.*

Here are some reminders about some C functions that appear in the question. `assert` (which is technically a macro) takes as an argument a boolean condition that is supposed to be true. If the condition is true nothing happens, and if the condition is false the program immediately stops with an error message. `malloc` takes a number of bytes as an argument, and allocates that much memory, returning a pointer to the allocated memory. The pointer returned by `malloc` should be passed to `free` when the program is done with it. `strcpy` and `strcat` both copy a string into a destination buffer in their first argument. The difference between them is that `strcpy` overwrites the buffer from the beginning, while `strcat` performs concatenation by copying its second argument after the end of the string already in the destination buffer.

4. (20 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) __**S**__ Library function to execute a string with a shell

   (b) __**C**__ System call to change user and group associated with a file

   (c) __**R**__ Exempt from all discretionary access-control checks

   (d) __**G**__ Virtual machine underneath a normal kernel

   (e) __**B**__ Allow-list-style mechanism to stop control-flow hijacking

   (f) __**K**__ x86-64 register pointing to the beginning of a stack frame

   (g) __**H**__ CPU mode where all memory is accessible

   (h) __**E**__ Specifying which inputs constitute an attack

   (i) __**Q**__ Subject to buffer overflow and format string bugs

   (j) __**O**__ Stores CPU registers in a memory buffer

   (k) __**J**__ Compilation mode where all code is position-independent

   (l) __**M**__ Added to password hash to conceal equality

   (m) __**D**__ Allows information to flow in only one direction

   (n) __**A**__ A value which, if overwritten, indicates an attack

   (o) __**L**__ x86-64 register pointing to the top of the stack

   (p) __**P**__ Architecture vulnerability related to, e.g., branch prediction

   (q) __**N**__ An isolated environment for untrusted code

   (r) __**T**__ CPU mode where page tables cannot be changed

   (s) __**I**__ Trusted Computer System Evaluation Criteria

   (t) __**F**__ Point where false-positive and false-negative rates are equal

   A. canary B. CFI C. `chown` D. data diode E. deny list F. EER G. hypervisor H. kernel mode I. Orange Book J. PIE K. `%rbp` L. `%rsp` M. salt N. sandbox O. `setjmp` P. Spectre Q. `sprintf(3)` R. superuser S. `system(3)` T. user mode