Paxson
Spring 2013

CS 161
Computer Security

Discussion 2

January 30, 2013

Question 1 ***Buffer Overflow Mitigations*** **(10 min)**

Buffer overflow mitigations generally fall into two categories: *(i)* eliminating the cause and *(ii)* alleviating the damage. In lecture, we saw memory-safe languages and proofs as examples for the first category. This question is about techniques in the second category.

Several requirements must be met for a buffer overflow to succeed. Each requirement listed below can be combated with a different countermeasure. With each mitigation you discuss, think about *where* it can be implemented—common targets include the compiler and the operating system (OS). Also discuss limitations, pitfalls, and costs of each mitigation.

(a) The attacker needs to overwrite the return address on the stack to change the control flow. Is it possible to prevent this from happening or at least detect when it occurs?

(b) The overwritten return address must point to a valid instruction sequence. The attacker often places the malicious code to execute in the vulnerable buffer. However, the buffer address must be known to set up the jump target correctly. One way to find out this address is to observe the program in a debugger. This works because the address tends to same across multiple program runs. What could be done to make it harder to accurately find out the address of the start of the malicious code?

(c) Attackers often store their malicious code in the same inside the buffer that they overflow. What mechanism could prevent the execution of the malicious code? What type of code would break with this defense in place?

Solution:

(a) **Stack Canaries.** A *canary* or *canary word* is a known value placed between the local variables and control data on the stack. Before reading the return address, code inserted by the compiler checks the canary against the known value. Since a successful buffer overflows needs to overwrite the canary before reaching the return address, and the attacker cannot predict the canary value, the canary validation will fail and stop execution prior to the jump.

As an example, consider the following function.

```
void vuln()
{
    char buf[32];
    gets(buf);
}
```

The compiler will take this function and generate:

```
/* This number is randomly set before each run. */
int MAGIC = rand();

void vuln()
{
    int canary = MAGIC;
    char buf[32];
    gets(buf);
    if (canary != MAGIC)
        HALT();
}
```

Limitations.

- Canaries only protect against stack smashing attacks, not against heap overflows or format string vulnerabilities.
- Local variables, such as function pointers and authentication flags, can still be overwritten.
- No protection against buffer *underflows*. This can be problematic in combination with the previous point.
- If the attack occurs before the end of the function, the canary validation does not even take place. This happens for example when an exception handler on the stack gets invoked before the function returns.
- A canary generated from a low-entropy pool can be predictable. In 2011 research showed that the Windows canary implementation only relied on 1 bit of entropy [3].

Cost. The canary has to be validated on each function return. The performance overhead is only a few percent since a canary is only needed in functions with local arrays. To determine whether to use the canary, Windows additionally applies heuristics (which unfortunately can also be subverted [1].)

(b) **Address Randomization.** When the OS loader puts an executable into memory, it maps the different sections (text, data/BSS, heap, stack) to fixed memory

locations. In the mitigation technique called *address space layout randomization* (ASLR), rather than deterministically allocating the process layout, the OS randomizes the starting base of each section. This randomization makes it more difficult for an attacker to predict the addresses of jump targets. For instance, the OS might decide to start stack frames from somewhere other than the highest memory address.

Limitations.

- Entropy reduction attacks can significantly lower the efficacy of ASLR [6]. For example, reducing factors are page alignment requirements (stack: 16 bytes, heap: 4096 bytes).
- Address space information disclosure techniques can force applications to leak known addresses (e.g., DLL addresses).
- Revealing addresses via brute-forcing can also be an effective technique when an application does not terminate, e.g., when a block that catches exceptions exists.
- Techniques known as *heap spraying* and *JIT spraying* [2] allow an attacker to inject code at predictable locations.
- Like the canary defense, ASLR also does not defend against local data manipulation.
- Not all applications work properly with ASLR. In Windows, some opt out via the `/DYNAMICBAS` linker flag [4].

Cost. The overhead incurred by ASLR is negligible.

(c) **Executable Space Protection.** Modern CPUs include a feature to mark certain memory regions non-executable. AMD calls this feature the NX (**n**o e**x**ecute) bit and Intel the XD (e**x**ecute **d**isable) bit. The idea is to combat buffer overflows where the attacker injects own code.

OpenBSD pioneered this technique in 2003 with W⊕X (write x-or execute), which marks pages as either writable or executable, but not both at the same time. Since Service Pack 2 in 2004, Windows features *data execution prevention* (DEP), an executable space protection mechanism that uses the NX or XD bit to mark pages, which are intended to only contain data, as non-executable.

Limitations.

- An attacker does not have to inject its own code. It is also possible to leverage existing instruction sequences in memory and jump to them. See part Question 2 for details.
- The defense mechanism disallows execution of code generated at runtime, such as during JIT compilation or self-modifying code (SMC).

- If code is loaded at predictable addresses, it is possible to turn non-executable into executable code, e.g., via system functions like `VirtualAlloc` or `VirtualProtect` on Windows [4].

Cost. There is no measurable overhead due to the hardware support of modern CPUs.

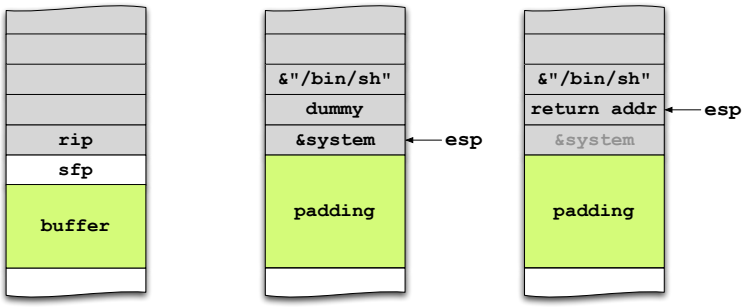
Question 2 ***Arc Injection*** **(10 min)**

Imagine that you are trying to exploit a buffer overflow, but you notice that none of the code you are injecting will execute for some reason. How frustrating! You still really want to run some malicious code, so what could you try instead?

Hint: In a stack smashing attack, you can overwrite the return address with any address of your choosing.

Solution: Rather than injection code, the main idea of *arc injection* is to inject data. It is a powerful technique to that bypasses numerous protection mechanisms, in particular executable space protection (Question 1c). By injecting malicious data that existing instructions later operates on, an attacker can still manipulate the execution path.

For example, an attacker can overwrite the return address with a function in `libc`, such as `system(const char* cmd)` whose single argument `cmd` is the new program to spawn. The attacker also has to setup the arguments (i.e., the data) appropriately. Recall that function arguments are pushed in reverse order on the stack before pushing the return address. Consider the example below, where an attacker overwrites the return address with the address of system (denoted by `&system`) to spawn a shell.



The first figure on the left is the stack layout before the attack. The second figure in the middle represents the state after having overflowed the buffer. Here, the return address is overwritten with `&system`. The value above is the location of the return address, from the perspective of `system`'s stack frame. But since the attacker plans on spawning a shell that blocks to take evil commands (e.g., `rm -rf /`), this value will never be used — hence any dummy value will suffice. The argument to `system` is the address of attacker-supplied data, in this case a pointer to the string `/bin/sh`.

Finally, the third figure displays the stack state after transferring control to `system`, which happened by popping `&system` into the program counter (and decrementing the stack pointer). At this point, the attacker can execute commands using the shell.

A more sophisticated version of arc injection is called *return-oriented programming* (ROP) [5]. It is based on the observations that the virtual memory space (which has the C library) offers many little code snippets, *gadgets*, that can be parsed as a valid sequence of instructions and end with a `ret` instruction. Recall that the `ret` instruction is equivalent to `popl %eip`, i.e., it writes the top of the stack into the program counter. The attacker does not even have to jump to the start of a function, any arbitrary location in the middle works as long as it terminates with a `ret`.

Shacham et al. showed that these small gadgets can be combined to perform arbitrary computation. In our above example, a basic combination of two gadgets would involve writing the starting address of the next gadget at the value of `dummy`. When the first gadget finishes, the next one is loaded by executing `ret`.

Setting up the stack is very tricky to get right manually, but the paper referenced above actually wrote a compiler to transform code from a language as expressive as C-like into mixture of gadgets to be pushed on the stack!

Question 3 ***Reasoning about Memory Safety*** **(10 min)**

Consider the following C code:

```
int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    if (issafe(s[j]))
    {
      s[i] = s[j];
      i++; j++;
    }
    else
    {
      j++;
    }
  }
  return i;
}

int issafe(char c)
{
  return ('a' <= c && c <= 'z') || ('0' <= c && c <= '9') || (c == '_');
}
```

We'd like to know the conditions under which `sanitize` is memory-safe, and then prove it. On the next page, you can find the same code again, but with blank spaces that you need to fill in (a-f). You don't need to prove the safeness of `issafe`.

Recall our proving strategy from lecture:

1. Identify each point of memory access
2. Write down precondition it requires
3. Propagate requirement up to the beginning of the function

HINT: Propagating the requirement up to the beginning of the function is more involved than in lecture. Here you need to reason about the properties that hold about the array indices after they are modified.

```
/* (a) requires:                                                    */
                 ------------------------------------------------
int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {


    /* (b) invariant:                                               */
                      ---------------------------------------------
    if (issafe(s[j]))
    {


      /* (c) invariant:                                             */
                        -------------------------------------------
      s[i] = s[j];
      i++; j++;


      /* (d) invariant:                                             */
                        -------------------------------------------
    }
    else
    {
      j++;


      /* (e) invariant:                                             */
                        -------------------------------------------
    }


    /* (f) invariant:                                               */
                     ----------------------------------------------
  }
  return i;
}
```

Solution:

Note: you might be able to develop the necessary invariants using simpler or more direct analytical approach. Here we strive to proceed very methodically using a large number of small steps, as that's the most clear and least error-prone process. Such care can become particularly important when attempting to reason about more complex code. But if you can work out proofs correctly in a more direct manner, that's fine. The bottom line is making sure you have a correct proof!

Since the goal is to prove memory safety, the first step is to identify points in the code that can violate this property, and write down the preconditions for these points of interest. This step establishes what we ultimately need to prove.

In this particular example, only the array accesses `s[i]` or `s[j]` can violate the memory safety property. Let's write down the preconditions for statements involving these memory accesses.

```
/* requires: s!= NULL && 0 <= j < size (s) */
issafe(s[j])
```

Note that we only wrote down the condition for this particular array access, ignoring everything else in the program. Similarly,

```
/* requires: s != NULL && 0 <= j < size (s) && 0 <= i < size (s) */
s[i] = s[j]
```

We wrote the above preconditions using compound relationals (like $a \leq b < c$). While more succinct, those are actually harder to reason about because they involve more than one condition, and during the reasoning process we may know that part of the compound holds, but not yet be sure we have the right form for the other part. So we rewrite them in expanded form:

```
/* requires: s!= NULL && 0 <= j && j < size (s) */
issafe(s[j])
...
/* requires: s != NULL && 0 <= j && j < size (s) && 0 <= i && i < size (s) */
s[i] = s[j]
```

Now, we need to propagate the *implications* of these requirements up to the function precondition. That is, given we need those preconditions to hold, what does that *ultimately* translate into in terms of the precondition for calling the function? When dealing with loops, this propagation step involves reasoning about code that comes *after* the memory accesses as well.

The procedure for propagation is to walk through the code writing invariants at each point in the code. *Recall that an invariant is what is guaranteed to be true of the state of variables whenever we reach that point in execution.* In developing these invariants, we will establish elements of them by imposing preconditions earlier in

the code's execution, ultimately up to the calling of the function itself. Doing so can take repeated passes through a set of *candidate* invariants, firming up each element in an invariant as we can reason it must hold, or perhaps adjusting it if we discover we didn't get it quite right the first time.

How do we construct the invariants? One point of confusion is: "for some invariants, isn't part of the invariant what we are trying to prove?" For example, how do we know what state of variables we can guarantee to hold at point (b) in this code? We start off with a framing of what we believe the invariant will look like, marking each uncertain element with "`?`", and then set about successively examining the invariants to determine which parts we can directly establish. Thus, we start off with parts marked *does it hold?*, and update those to *it holds* as we work through the code.

Let's now walk through the code and illustrate the above process.

Step 1:

(a) To begin, we don't have any clear picture yet regarding the precondition for the function, so this is simply "`?`". We will come back to this later once we have worked out what needs to be propagated.

(b) The loop condition ensures `j < n`. This will clearly *always* hold at this point.

Note that while the loop initialization ensures that the first time executing we have `i = j = 0`, that does not necessarily hold for subsequent iterations (indeed, we would not expect it to), so we don't include that fact in our draft invariant here.

So for this part, for our first step we have the precondition required by the next statement, marked as uncertain, and the fact ensured by the loop condition, which is not uncertain.

(c) The preceding `if` statement doesn't modify `j`, so we can include the loop condition at this point too. The rest of our candidate invariant here comes from the precondition for the assignment.

(d) Here we take the invariant from above and update it to reflect that `i` and `j` have been incremented. (There's actually a subtle point here: incrementing can cause variables to overflow. Therefore, what we've written down here won't in fact necessarily hold. We return to this point later, as an example of how we can sometimes have to fix up a candidate invariant when we discover we can't prove it.)

(e) The reasoning here is similar to that for (d), except only `j` has been incremented.

(f) At this point, we take what we can logically establish given that we know *either* (d) holds or (e) holds, but we don't know *which* one. Thus, our candidate invariant is the minimum that follows from both of them, giving us something that is true regardless of which of the if-else branches is taken. (This is why our

candidate invariant for (f) doesn't make mention of constraints on `i`, since (e) doesn't provide any such constraints.)

```
/* (a) requires: ?                                                  */
                 ---------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= j? && j < size(s)? && s != NULL? */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i? && i < size(s)?
                        && 0 <= j? && j < size(s)? && s != NULL? */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 < i? && i <= size(s)?
                        && 0 < j? && j <= size(s)? && s != NULL? */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 < j?
                        && j <= size(s)? && s != NULL? */
    }

    /* (f) Invariant: j <= n && 0 < j?
                      && j <= size(s)? && s != NULL? */

  }
  return i;
}
```

Step 2: Right now there are a lot of "?"s. We can do a bit of initial simplification regarding `s != NULL?`. Since the code never alters `s` itself, the only way to ensure that `s != NULL` holds is that it's a precondition for calling the function. That gives us this simple update:

```
/* (a) requires: s != NULL && ?                                    */
                 ---------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= j? && j < size(s)? && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i? && i < size(s)?
                        && 0 <= j? && j < size(s)? && s != NULL */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 < i? && i <= size(s)?
                        && 0 < j? && j <= size(s)? && s != NULL */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 < j? && j <= size(s)? && s != NULL */
    }

    /* (f) Invariant: j <= n && 0 < j? && j <= size(s)? && s != NULL */

  }
  return i;
}
```

That is, we added to the function's precondition, and removed the "?"s around the `s != NULL` parts of the invariants, since given the precondition, we're now confident that those hold.

Step 3: The next simplification is to observe that we often want to establish conditions like `0 <= i?` or `0 < i?`. The first of these follows immediately because `i` and `j` are type `size_t`, so we can mark those as resolved (remove the "?").

Note that that observation does not quite allow us to do away with the ?", because simply the fact that `i` is of type `size_t` does not guarantee that it is strictly larger than 0. Indeed, even if it *was* larger than zero before being incremented, it might overflow and *become* zero. That observation suggests that some of our original can-

didate invariants had flaws (things we might not be able to prove). Accordingly, we decide to *relax* those invariants that have conditions like to `0 < i?` to instead have `0 <= i?`. These latter then can be changed to `0 <= i` because of the observation that `i`'s type is `size_t`.

In general, this notion of making an invariant more relaxed is an important reasoning technique to keep in mind. Sometimes the immediate constraint implied by a statement is more narrow than what we need for our overall reasoning. Note that "relax" means "make broader"; you have to ensure that relaxing the constraint has not undermined the original preconditions for memory safety.

Another note: had the variables instead been of type `int`*, then the reasoning here would have been more difficult. Indeed, we would need to resort to* induction *to reason about requirements that ensure that the condition holds. We will see an example of using induction below in Step 8.*

This leads then to:

```
/* (a) requires: s != NULL && ?                                    */
                 ----------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= j && j < size(s)? && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i && i < size(s)?
                        && 0 <= j && j < size(s)? && s != NULL */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 <= i && i <= size(s)?
                        && 0 <= j && j <= size(s)? && s != NULL */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */
    }
```

```
    /* (f) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */

  }
  return i;
}
```

Step 4: When we inspect the above, what's left are questions about the relationship between `i` and `j` to `size(s)`. This part requires some thought. In particular, we have some candidate invariants inside the loop that want to provide conditions on `i`, but the candidate invariants at the top of the loop and at its bottom do *not* discuss `i`. That's going to make it hard to reason about properties of `i` inside the loop. So we extend the other candidate invariants to include information about `i` as well, by propagating the conditions we need inside the loop. This gives us:

```
/* (a) requires: s != NULL && ?                                    */
                 ----------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= i && i < size(s)?
                      && 0 <= j && j < size(s)? && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i && i < size(s)?
                        && 0 <= j && j < size(s)? && s != NULL */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 <= i && i <= size(s)?
                        && 0 <= j && j <= size(s)? && s != NULL */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 <= i && i < size(s)?
                        && 0 <= j && j <= size(s)? && s != NULL */
    }
```

```
    /* (f) Invariant: j <= n && 0 <= i && i < size(s)?
                      && 0 <= j && j <= size(s)? && s != NULL */

  }
  return i;
}
```

This change may strike you as obvious, and you may be wondering why we didn't include these statements about `i` in those other candidate invariants from the start. The reason is that *initially* we didn't have any particular requirements for memory safety at those points that directly needed a precondition on `i`. It's only as we try to establish properties on `i` inside the loop that we now see we have to reason about `i` at the loop's beginning and end. (Including `i` from the get-go would certainly be a reasonable short-cut to take, as long as you're always careful about verifying all of the steps in your reasoning.)

Step 5: Now we get to the crux of it. In (b), we have an invariant that we want to know always holds when beginning a new loop iteration. To repeat it:

```
(b) Invariant: j < n && 0 <= i && i < size(s)?
               && 0 <= j && j < size(s)? && s != NULL */
```

How can we establish that the "`?`" will hold? For `i`, it's not clear how to proceed, but we note that we have some additional information already established about `j`, namely `j < n` (first clause in the invariant). So if we happen to have `n <= size(s)` then it logically follows that `j < size(s)`.

Here we can inspect the code and note that `n` never changes. Thus, if we'd like `n <= size(s)`, that will need to be a precondition for calling the function.

This gives us:

```
/* (a) requires: s != NULL && n <= size(s) && ?                */
                 ---------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= i && i < size(s)?
                      && 0 <= j && j < size(s) && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i && i < size(s)?
```

```
                        && 0 <= j && j < size(s) && s != NULL */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 <= i && i <= size(s)?
                        && 0 <= j && j <= size(s) && s != NULL */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 <= i && i < size(s)?
                        && 0 <= j && j <= size(s) && s != NULL */
    }

    /* (f) Invariant: j <= n && 0 <= i && i < size(s)?
                      && 0 <= j && j <= size(s) && s != NULL */

  }
  return i;
}
```

Step 6: All that's left is that pesky `i < size(s)?`. Here we need to bring a bit of reasoning to the table: why do we *think* it'll be the case that `i` never exceeds `size(s)`? Well, because it looks like `i` never exceeds `j`, and we know (now) that `j` never exceeds `size(s)`. So we *add to our candidate invariants* the notion that i never exceeds j, which so far isn't proven:

```
/* (a) requires: s != NULL && n <= size(s) && ?                */
                 ---------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= i && i < size(s)? && i <= j?
                      && 0 <= j && j < size(s) && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i && i < size(s)? && i <= j?
                        && 0 <= j && j < size(s) && s != NULL */
```

```
        s[i] = s[j];
        i++; j++;

        /* (d) Invariant: j <= n && 0 <= i && i <= size(s)? && i <= j?
                          && 0 <= j && j <= size(s) && s != NULL */
      }
      else
      {
        j++;

        /* (e) Invariant: j <= n && 0 <= i && i < size(s)? && i <= j?
                          && 0 <= j && j <= size(s) && s != NULL */
      }

      /* (f) Invariant: j <= n && 0 <= i && i < size(s)? && i <= j?
                        && 0 <= j && j <= size(s) && s != NULL */

    }
    return i;
  }
```

Step 7: At this point, it's helpful to review some of the internal logic for this new condition. Let's assume for the moment that (b) has been fully established. Then (c) follows immediately, because there are no changes to `i` or `j` at the intervening step. (d) also follows immediately, since both `i` and `j` are changed by the same amount.

(e) is trickier. `j` has increased, but `i` hasn't. Normally we might think well surely then if we had `i <= j` going into that increment, we will have it after doing the increment of `j`. But here we have to consider *overflow*, always a subtle part of reasoning about code. The key insight is that for `j` to overflow, it must have been equal to the largest expressible unsigned value just prior to the overflow. However, another part of (b) (which remember we're assuming for now has been fully established) tells us `j < size(s)`. Since the size of a memory region can't exceed what's expressible in a `size_t` variable, that condition ensures that `j` is *less* than the largest expressible unsigned value. Therefore, overflow can't occur; and therefore, we will have `i <= j` at (e) if (b) has been fully established.

Given that reasoning for (d) and (e), we know we then also have it at (f).

Step 8: Almost done! Okay, how do we establish that `i <= j` indeed holds at (b)? When reasoning about invariants at the beginning of loops, we can need to employ *induction*. The *base case* is the first entry into the loop. Here, we can use the loop's initialization, which is that both `i` and `j` are 0. For that, we have `i <= j` — so far, so good.

The inductive step has two parts. First, we need to show that if we assume that the

invariant at the beginning of the loop holds, then given that we can show that the invariant at the end of the loop (f) holds. It should be straightforward for you to confirm that this is indeed the case. Second, we need to show that if we assume the invariant at the end of the loop *plus* the loop condition, then we can establish that the invariant at the beginning of the loop holds. That is, we assume both:

```
/* (f) Invariant: j <= n && 0 <= i && i < size(s) && i <= j
                  && 0 <= j && j <= size(s) && s != NULL */
```

and `j < n`. Again, it should be clear upon inspection (including the precondition that `n <= size(s)`) that these two when combined indeed give us:

```
/* (b) Invariant: j < n && 0 <= i && i < size(s) && i <= j
                  && 0 <= j && j < size(s) && s != NULL */
```

That completes the reasoning. We've established by induction that `i <= j` holds in (b), and from that and `j < size(s)` we have also established `i < size(s)`. So we don't need any more preconditions for the function, and the final answer is:

```
/* (a) requires: s != NULL && n <= size(s)                    */
                 ---------------------------------------------

int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) Invariant: j < n && 0 <= i && i < size(s) && i <= j
                      && 0 <= j && j < size(s) && s != NULL */
    if (issafe(s[j]))
    {

      /* (c) Invariant: j < n && 0 <= i && i < size(s) && i <= j
                        && 0 <= j && j < size(s) && s != NULL */
      s[i] = s[j];
      i++; j++;

      /* (d) Invariant: j <= n && 0 <= i && i <= size(s) && i <= j
                        && 0 <= j && j <= size(s) && s != NULL */
    }
    else
    {
      j++;

      /* (e) Invariant: j <= n && 0 <= i && i < size(s) && i <= j
```

```
                          && 0 <= j && j <= size(s) && s != NULL */
    }

    /* (f) Invariant: j <= n && 0 <= i && i < size(s) && i <= j
                      && 0 <= j && j <= size(s) && s != NULL */

  }
  return i;
}
```

Phew!

We can summarize the overall reasoning strategy using the following template, which is elaborated a bit from the one presented in lecture:

(1) Identify any memory accesses that could have safety issues.

(2) Write down preconditions for these.

(3.1) Write down *candidate invariants* that will satisfy these preconditions.

(3.2) Repeatedly identify any elements of those invariants that you can directly reason about and mark them as no longer uncertain. This may involve adding preconditions to the function.

(3.3) For what remains, determine what sort of constraints one might *add* to the candidate invariants to enable proving the missing pieces, and also possibly what conditions to *relax* (make broader). This may enable going back to 3.2.

(3.4) Use induction to indirectly reason about remaining elements. From here, one can go to 3.2 or 3.3.

(4) Once you've resolved all of the candidate invariants, you're done.

A final note: do not hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.

References

[1] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038.

[2] Dionysus Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf, February 2010.

[3] Matthew "j00ru" Jurczyk and Gynvael Coldwind. Exploiting the otherwise non-exploitable. http://j00ru.vexillium.org/?p=690, January 2011.

[4] Matt Miller. On the effectiveness of DEP and ASLR. http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx, December 2010.

[5] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[6] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.