Paxson
Spring 2011

# CS 161 Computer Security

Discussion 1

## January 26, 2011

**Question 1** ***Buffer Overflow Mitigations***

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 coe 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, the OS checks the canary against the known value. Because a successful buffer overflows needs to overwrite the canary before reaching the return address, but the attacker cannot predict its value, the canary validation will fail and invalidate the return address.

As an example, consider the following function.

```
void vuln()
{
    char buf[n];
    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[n];
    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 *underruns*. 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. Recent research showed that the Windows implementation only relies on 1 bit of entropy [**?**].

**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 [?].)

(b) **Address Randomization.** Precisely, this mitigation technique is called *address space layout randomization* (ASLR). When OS loader puts an executable into memory, it maps the different sections (text, data/BSS, heap, stack) to fixed locations. Rather than deterministically allocating the process layout, the idea behind ASLR to randomize the starting base of the sections to make it more difficult for an attacker to prediction 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 [?]. 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* [?] 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 [?].

**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 jumpt 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 [?].

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

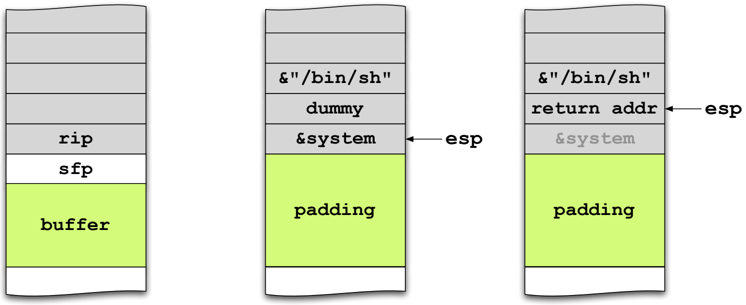
**Question 2** ***Arc Injection***

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 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 happend 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) [?]. 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***

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. Find the blank space labelled `requires` and fill it in with the precondition that's required for `sanitize` to be memory-safe. (If several preconditions are valid, you should list the most general precondition under which it is memory-safe.)

Also, on the next page fill in the four blanks inside `sanitize` with invariants, so that *(i)* each invariant is guaranteed to be true whenever that point in the code is reached, assuming that all of `sanitize`'s callers respect the precondition that you identified, and *(ii)* your invariants suffice to prove that `sanitize` is memory-safe, assuming that it is called with an argument that satisfies the precondition that you identified.

Here is the same C code again, this time with space for you to fill in the precondition and four invariants.

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

    /* invariant: ___________________________________________ */
    if (issafe(s[j]))
    {
      s[i] = s[j];
      i++; j++;

      /* invariant: _________________________________________ */
    }
    else
    {
      j++;

      /* invariant: _________________________________________ */
    }

    /* invariant: ___________________________________________ */
  }
  return i;
}

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

**Solution:**

Since the goal is to prove memory safety the key is to identify points in the code where this property can get violated. In this particular example, only the array accesses `s[i]` or `s[j]` can violate our property. In another program, the function `fgets` could be our point of interest. This first step is key but easy to forget in a some corner code point.

Now, we can write down the precondition for these points of interest, in our case all the array accesses. This is simple. Here's an example:

```
/* 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, conditions can be written for all the other points in the program.

The intuition for the rest of the proof construction is that we want to go up one line of code at a time, modifying our precondition appropriately. In the absence of loops, this is usually trivial. For example, consider the following modified function:

```
int sanitize(char s[], size_t n)
{
  int j=0;
/* requires: s != NULL && -1 <= j < size(s) - 1 */
  j++;
/* requires: s != NULL && 0 <= j < size (s) */
  if (issafe(s[j]))
    printf("safe!");
}
```

The last precondition is the same as our example, and we have just gone up line by line appropriately modifying the program. Try and write down the precondition for the topmost line. It should be

`requires: s != NULL && 1 < size(s).`

The precondition for the top most line is also the precondition for the function.

With loops, things are more difficult. This isn't surprising: loops are what make computers powerful after all. Key to writing proofs for loops is coming up with a loop invariant.

A loop invariant is a property that is true before the start of the loop and remains true in each iteration of the loop as well as at the end of the loop. This is the non-mechanistic part: writing invariants requires creativity and, like real mathematical proofs, there is a certain art and experience involved. But for this question in particular, the invariant is not that hard, viz. `0 <= i <= j <= n`. Take a look at the code and make sure you understand why this is a correct invariant for the loop. In particular, the `j <= n` is less or equal and not lesser than, since the loop invariant has to be true at the end of the loop too.

Combined with our preconditions for array access memory safety, we get the safety precondition for the loop as

```
s != NULL && 0<= i <= j <= n <= size(s).
```

An interesting exercise here is to change the loop condition from `j < n` to `j < 3`. How does the loop invariant and the loop precondition change?

Now we can proceed with our mechanistic work and go back up one more line of code to get the precondition for the topmost line of code. We get

```
s != NULL && 0 < n <= size(s).
```

The initialization of `i` and `j` to zero means that we don't have to worry about the value of `i` and `j` before the execution of the first line of code.

We are done! The precondition for the topmost line of code is also the precondition for the function. The complete annotated program, with the other invariants is below:

```
/* (a) requires: s != NULL && 0 < n <= size(s) */
int sanitize(char s[], size_t n)
{
  size_t i = 0, j = 0;
  while (j < n)
  {
    /* (b) invariant: s != NULL && 0 <= i <= j <= n <= size(s) */
    if (issafe(s[j]))
    {
      s[i] = s[j];
      i++; j++;
      /* (c) invariant: s != NULL && 0 < i <= j <= n <= size(s) */
    }
    else
    {
      j++;
      /* (d) invariant: s != NULL && 0 <= i < j <= n <= size(s) */
    }
    /* (e) invariant: s != NULL && 0 <= i <= j <= n <= size(s) */
  }
  return i;
}
```

Note how the invariant changes after the increments to `i` and `j`.

Another exercise for elucidation is to replace all `s[i]` and `s[j]` with, say, `s[3]`. How does the function precondition change?

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.