Software Vulnerabilities

August 31, 2011 / CS261 Computer Security

Review paper discussion

Student interests from the review paper were:

- trampolining
- heap smashing
- more recent attacks

Trampolining

This term is seen in two contexts:

- compilers sometimes generate code stubs to be used for later. These can be used in arc injection[1] (return-to-libc) or return oriented computing (see p. 9); or,
- independent of arc-injection: suppose that the buffer's address is not known, but is stored in a register or on the stack. The exploit first jumps to code at a different, known location, which subsequently manipulates the register/stack values to jump to the buffer's location.

  This is explained in the paper (albeit without example code).

Heap smashing

This is hinted at in the paper but not fully explained.

malloc/free

malloc maintains a double-linked list of allocated blocks. Whenever malloc returns the starting address of a block of memory (indicated by the arrow), it will have also have allocated extra memory preceding this block, for storing the addresses of the “previous” and “next” pointers (Figure 1).

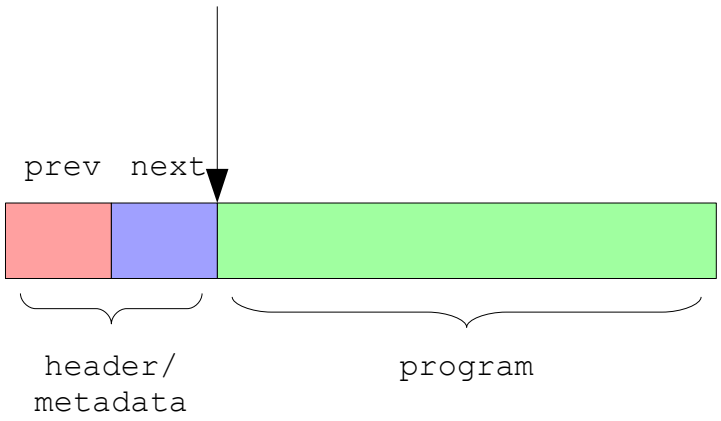


Figure 1. A single malloc'ed region.

1 The term “arc injection” is rarely used outside of the paper.

Figure 2 shows three nodes of the linked list of allocated regions, as maintained by malloc.

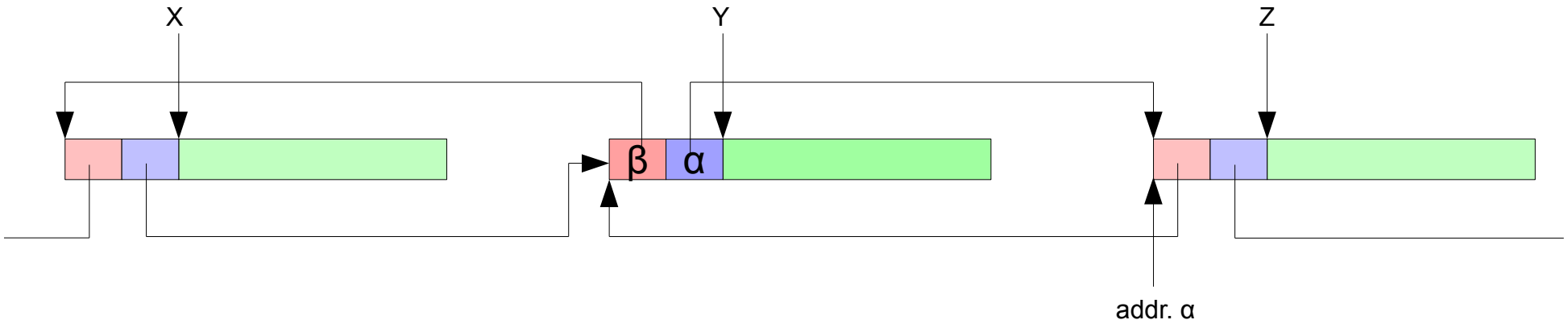


Figure 2. Three nodes of the malloc linked list.

When the operation

```
free (Y);
```

is performed, malloc deletes the node from the linked list:

```
p = Y - 8; // assuming 32-bit addresses
q = p->next;
p->next->prev = p->prev;
p->prev->next = q;
```

If `p->next` contains α and `p->prev` contains β (as per the figure), then the highlighted line of code above is equivalent to:

```
*α = β;
```

This can be exploited by:

1) Overwriting the header of Y (by overrunning the previous malloc'ed buffer, `X`[2]), with α and β set to any address and value of the attacker's choice respectively
2) When `free` is called: `*α = β;`

This primitive can be used for many attacks, such as:

- arc injection, if `α` was a function pointer
- modifying the return address
- changing any local variable e.g., canaries (see p. 6), authentication flags

Double freeing

Besides the potential to crash the program, this could also be a security vulnerability e.g.,

1) Y is `free`'d
2) X is reallocated to be larger, and now extends up to or beyond Y. The attacker can now change `α` and `β`
3) As per before, the arbitrary write occurs when Y is `free'd`

2 There are many other memory safety attacks which could be used instead, such as format string vulnerabilities.

Defenses

Classification

Two examples of defenses are:

- `free`/`malloc` could be replaced with a sandbox/boundary check. There could be a hash stored, or a separate data structure entirely.
- Use Java.

More generally, defenses can be classified as being applicable to legacy code or new code (Table 1).

Table 1. *Defenses.*

Legacy C/C++ code	New code
• W^X (see p. 9) • ASLR: malloc chooses random locations, the stack starting location is random etc.: this makes it hard to predict addresses (see p. 10) • more cautious `free`/`malloc` • guard pages: one unallocated page after each `malloc`, so that writing past the end will result in a segfault • canaries	• Use newer languages that have “memory safety” e.g., Java

Canaries

Design

When a function is called, a stack frame is created, containing local variables and arguments, as well as compiler added information, such as the return address (Figure 3).

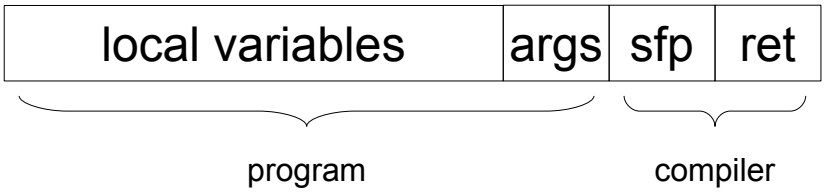


Figure 3. Stack frame.

A canary is a (usually) randomly chosen value stored between the local variables and compiler added information (Figure 4).

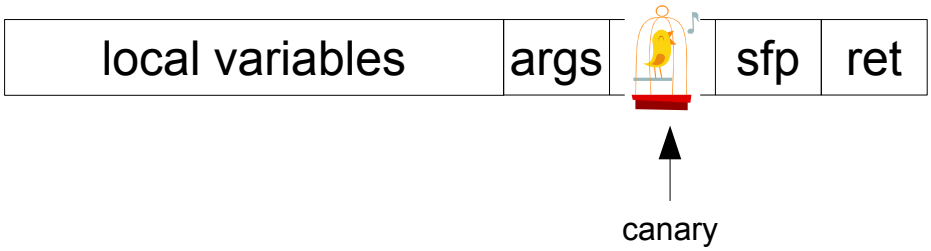


Figure 4. Stack frame with a canary.

The compiler modifies the program to push the canary into the stack frame, and check it before the function returns, on the assumption that if the return address has been overwritten by a buffer overflow in the local variables, the canary would also have been modified.

Stack canaries are used in many products such as StackGuard, and recent versions of gcc and Microsoft Visual C++. It is also possible in principle, though uncommon, to have heap canaries (added and checked by `free/malloc`).

Limitations

The canary approach assumes that the buffer overflow results in a contiguous overwrite (e.g., the `gets` function). With heap smashing, it is possible to overwrite the return address without affecting the canary.

Even if a contiguous overwrite occurs, the canary is still imperfect, because it does not detect buffer overflows immediately, thus code can be run before the canary check occurs.

As the reference canary is often a global variable i.e.,

1) local canary (in the stack frame) = global canary
2) perform function code
3) check local canary against global canary, prior to returning

it is vulnerable to rewriting (see also paper). For example, if the buggy code was of a particular form:

```
vuln () {
    ...
    // stack overflow
    ...
    *p = i;
    ...
}
```

and the local variables were laid out similarly to Figure 5:

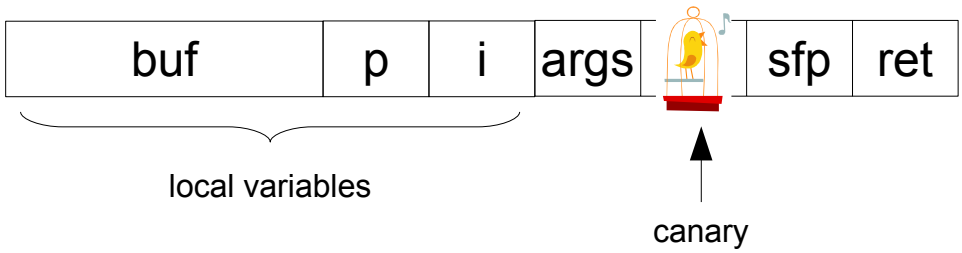


Figure 5. Vulnerability if code contains a buffer overflow, plus `*p = i;`.

then the code could be exploited by overflowing the buffer to change:

1) p, to the address of the reference/global canary
2) i, to an arbitrary value, say, 3441

   The operation `*p = i;` will set the global canary to 3441.
3) canary (in the stack frame), to the same value as i
4) return address, to taste

In this case, the canary just adds an extra level of complexity for attackers, requiring the code to have a stack overflow plus an additional condition.

The canary approach could be improved by:

- requiring arbitrarily more copies of the canaries and/or return addresses
- checking the canary after every write, or after every variable length buffer operation (e.g., strcpy)

but these could adversely affect performance, and only make attacks harder, not impossible.

Finding the canary

The buffer overflow exploit above assumed that the address of the reference/global canary was known to the attacker. As global variables are often allocated statically, and compilers are fairly deterministic, it would be possible to identify the location of the canary by compiling the code on a

system running the target architecture, or by obtaining a copy of the binary.

Canaries could also be stored directly in the code for each function, either:

- a constant value e.g., a hard-coded check against 9441: this is not very useful because multiple users would have the same binary, and therefore the same canary; or,
- with code that modifies the canary on every execution: this is complicated and could even introduce contain its own vulnerabilities.

In both cases, an attacker could trace the code.

In general, the canary could be defeated if there are 2 bugs: one which reads past a buffer (and displays the output to the attacker, such as via a network port), and another which writes past a buffer. As with the previous canary enhancements, it is possible to make the attacker work harder, and need more prerequisites, but it is not full proof.

Nonetheless, canaries are often used because they are relatively cheap and defend against some bugs.

Categories of Defenses

Defenses can be categorised as:

- run-time: these instrument the code or modify the system; or,
- static: compile time checks

All the legacy C/C++ code defenses listed in Table 1 are run-time. Java is actually a hybrid: it has run-time bounds checking, but also some static analysis to optimise away unnecessary checks.

Modifying the C compiler to include run-time checks is non-trivial because:

- at compile-time, the size of malloc'ed regions might not be known: it would be necessary to store states
- it is not just `array [index]` operations which need to be checked, but also `*ptr`. Note that `p` could be the start or middle of an array, so checking would be tricky or expensive.

The biggest challenge is performance (typically, there is a 2-4x slowdown), which is dealt with in next week's reading on “baggy bounds checking”.

Java's run-time bounds checking is simplified because:

- it does not have pointer data types
- every array has an associated length
- it is type-safe (whereas in C, a pointer could have been casted from another variable)

Run-time Defenses for Legacy Code

1) Non-executable stack

This was one of the first defenses to be introduced.

Pages are marked as non-executable. This prevents basic stack smashing attacks, in which the return address was overwritten with the address of a payload in the stack.

However, a non-executable stack does not prevent:

- return-into-libc attacks
- more generally, replacing the return address with the address of existing code
- rewriting flags and other local variables
- (perhaps the simplest) overwriting the return address to point to a payload in the heap

2) DEP / NX / W^X[3]

This is a refinement of the non-executable stack approach: rather than blacklisting the stack, all areas of memory other than the code segment are rendered non-executable.

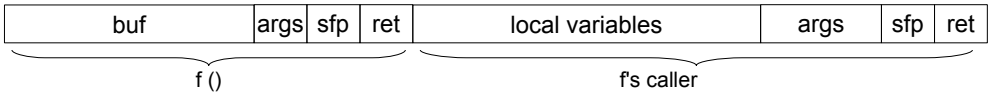
Depending on the architecture, it can be very difficult to overwrite the NX bit, as it would require modifying kernel space (thus presupposing that arbitrary system calls can be made by the attacker).

This does not protect against return-into-libc attacks, or indeed, arranging to call any function in the address space i.e., the attacker cannot introduce new code, but can repurpose existing code. A generalisation of return-into-libc is “return-oriented computing”.

Return-oriented computing

As previously illustrated in Figure 3, a function's arguments are provided in its stack frame. Figure 6 shows the stack frames for a function and its caller.

Figure 6. Stack frames for a function f () and its caller.



By overflowing `buf`, the return address of `f ()` could be replaced with that of, say, `execve`; furthermore, the stack frame of f's caller could be replaced with a suitable stack frame (i.e., correct arguments for) `execve`. The return address in `execve`'s stack frame could also be modified to that of another function, with an appropriate stack frame created for it as well ...

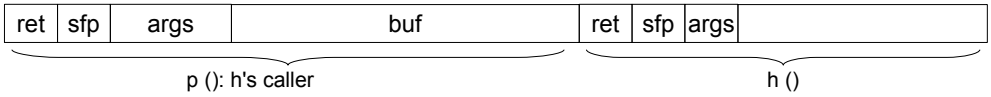
3 This is variously known as Data Execution Protection (DEP; Microsoft), No Execute (NX; x86 page table permission bit), or W xor X (W^X; BSD);

Return-oriented computing involves chaining together many short functions[4]; experimentally, it has been shown[5] that there are often enough "gadgets" to be Turing complete.

Stack conventions

In the attack above, it was assumed that the return address is placed after the local variables, and that the stack frame for f's caller appeared after f's (Figure 6). What if these conventions were reversed (Figure 7)?

Figure 7. Stack frames with conventions reversed.



In the following code, the `gets` in `h ()` is exploitable:

```
p () {
    char buf [512];
    h (buf);
}

h (char* b) {
    ...
    gets (b);
}
```

The main observation is that `gets ()` need not operate on a local variable. The `gets ()` stack frame could also act in place of `h ()`'s[6].

It would be interesting to quantify the relative vulnerability of such a stack arrangement, and the possible compatibility issues.

3) Separate stack

Not discussed.

4) Separate stack, heap metadata

Not discussed, other than that this is not widely deployed.

5) ASLR[7]

Address-space layout randomization involves modifying the beginning point of the heap, stack, and code segments.

NOP sleds

NOP sleds consist of many no operation instructions followed by the malicious code (Figure 8).

4 Offsets can be identified by trawling through the code
5 http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html
6 i.e., the return address of the gets stack frame could be overwritten.
7 See also: Active Straight Leg Raising; Age, sex, location, race

Figure 8. NOP sled.

By making a large sled (e.g., 2^{28} NOPs), a randomly chosen return address on a 32-bit system would have a 1 in 16 chance of executing the target code.

It is also common to employ “heap spraying”, especially against web browsers: using JavaScript to allocating as many memory objects (including NOP sleds) as possible to maximise the odds of a successful jump.

Limited random locations

Memory cannot be allocated at entirely arbitrary locations because of, for example, page alignment requirements.

On 32-bit Vista, global, heap, and stack variables only have 8, 5, and 14 bits of randomness respectively, making it relatively easy to guess addresses.

On 64-bit systems, ASLR is much more effective against heap spraying or guessing.

W^X + ASLR

Whereas W^X alone or ASLR by itself can each be attacked, the industry standard of W^X + ASLR is fairly robust, although not perfect. It has only a modest performance penalty, but has limitations that:

- not all programs enable W^X, because they need the ability to write and execute: for example: the JavaScript JIT compiler, web browsers (a big attack surface), Flash, Java JVM
- a small amount of code is non-randomised

Note: on embedded systems, performance and memory limitations often preclude using W^X and ASLR.

Conclusions

Buffer overflows

- While canaries are used, automatic bounds checking is better.
- The set of bugs is frequently revised. If there is memory corruption, assume that an attack is possible.
- Putatively good defenses, for which the designer “couldn't think of a way” to circumvent them, turned out to be wrong.

Security

- It is important to understand the nature of the problem in order to evaluate solutions.
- Security is an arms race between attackers and defenders.
- Security researchers are extremely cautious (perhaps paranoid).
- “Looks good to me” does not mean it is ok; it is preferable to have an affirmative argument.

Acknowledgements (external)

- Images from Microsoft (http://office.microsoft.com)
  - canary: MC900434593
  - trampoline: MC900089478
  - stack: MC900310532
- malloc discussion: patent #7028023
- Acronyms from http://acronyms.thefreedictionary.com
- Ads from Google (http://www.google.com/)