September 1: Memory safety, Buffer Overflows attacks

Scribe: Katia Patkin

1 Requirements

- System software written in an unsafe language (C), exposes raw pointers to the developer.
- Architectural layout of data.

```c
void login()
{
    char buf[16];
    int authenticated = 0;
    [other local var]
    gets(buf);
    ...
}
```

Attacker can:
- Set data (e.g. authentication bit).
- Get control of program flow, run with process privilege.

2 Types of Buffer Overflow

1. Stack Smashing: Attacker overwrites return address and points to attacker supplied code.

2. Arc injection: Attacker overwrite return address to points to existing code.

Example: return to libc
```c
void system(char * arg)
{
    check_validity(arg);
```
R = arg;
    target:
        exec1(R,...)
    }

Steps:
(a) Set return address to target.
(b) Ensure R (system register) points to attacker code (based on vulnerable program logics
    registers are reused)

3. **Pointer subterfuge:** Attack exploiting pointer overwrite.

4. **Heap Smashing:**

```c
int main(int argc, char* argv)
{
    p = malloc(1024);
    q = malloc(1024);
    strcpy(p, argv[1]);
    ...  
}
```

Simplified heap model:

[*] Upon block free (size=0) heap manager sets previous pointer to the next pointer.

Steps:
(a) Overwrite heap block such that previous pointer (α) points to return address, next
    pointer (β) points to attacker’s code and size=0.
(b) Heap manager frees block set location at α to point to β.

⇒ return address points to attacker code.

This attack is not very common, because the memory layout is less predictable and it is a
more complicated attack.
3 Fixes


2. Build tools that help programmers find bugs:

   Example:

   ```c
   void foo (int * p)
   {
       int offset;
       int* z = p + offset;
       ...
   }
   ```

   Static checker: Checks that `offset` is not initialized. `offset` hence can get any value, which means pointer could point to anything. Cons: hard to find all bugs.

3. Use a memory-safe language:

   Cons:

   - Not good for performance.
   - There is legacy code.
   - Not suitable for writing low-level code.

4. Bounds checking:

   - Canaries: Modifies source code

     Compiler places canary (random value) before local variables upon entry in function and checks before return.

     ```text
     return address
     saved bp
     canary
     buf[15]
     ...
     buf[0]
     ```

   - Electric fences:

     Object is followed by a guard page. Any access to guard page triggers page fault.

     ```text
     Guard page
     object
     ```

     Cons: takes a lot of memory space, can be used for DoS attacks.

   - Baggy bounds:

     Goal: to check that the pointers are in range.

     Example:
\begin{verbatim}
char x[1024];
char* y = &x[107];
y+2124 . . .
\end{verbatim}

Check for pointer arithmetic that it is in bound.
How: For a pointer p' that is derived from p. p' should only be dereferenced to access memory that belongs to p.

4 Fat pointers

Each pointer holds bound information:

<table>
<thead>
<tr>
<th>base</th>
<th>end</th>
<th>current address</th>
</tr>
</thead>
</table>

Cons:

- Performance overhead: for every pointer dereference, check bounds.
- Memory overhead: every 32-bit pointer is now 96-bit pointer.
- Incompatible with existing binaries.

5 Baggy bounds

Use data structures to keep bounds of each pointer.

Interpose on two operations:

1. pointer arithmetic:
   \begin{verbatim}
   char* q = p + 256
   \end{verbatim}
   Needed to check pointer provenance (which pointer it was derived from)

2. pointer dereference:
   \begin{verbatim}
   char p[256];
   \end{verbatim}
   Needed because in arithmetic intermediate value might be out of bound.

Implementation:

1. Align and allocate in the power of 2. Ex.: \texttt{malloc(44) \to 64}.
2. Express size of pointer as \(\log_2(alloc\ size)\).
3. Store pointer to size in a linear array.
4. Allocate memory at slot granularity (16 bytes for Baggy).
Example:
\( p = \text{malloc}(16) \rightarrow \text{alloc size} = 16, size = 4, slot = 1 \rightarrow \text{table}[p\text{slot size}]=4. \)
\( p = \text{malloc}(44) \rightarrow \text{alloc size} = 64, size = 6, slot = 4 \rightarrow \text{table}[p\text{slot size}]=6, \)
..., \text{table}[p\text{slot size}]=6

Check \( p' \) is in the bound of \( p \):

C code:
\( p' = p + i \)

Bounds check:
\( size = 1 \ll \text{table}[p \gg \log(slot size)] \)
\( base = p \& (size -1) \)
\( base \leq p' < base + size \)