Variable Initialization

- Some languages let you declare variables without specifying their initial values
- And let you use them without initializing them
  - E.g., C and C++
- Why is that a problem?

Variable Initialization

- Some languages let you declare variables without specifying their initial values
- And let you use them without initializing them
  - E.g., C and C++
- Why is that a problem?

A Little Example

```
main()
{

foo();
bar();
}

foo()
{
        int a;
        int b;
        int c;

        a = 11;
        b = 12;
        c = 13;
}
```

```
bar()
{
        int aa;
        int bb;
        int cc;

        printf("aa = %d\n",aa);
        printf("bb = %d\n",bb);
        printf("cc = %d\n",cc);
}
```

What's the Output?

```
lever.cs.ucla.edu[9]./a.out
aa = 11
bb = 12
cc = 13
```

- Perhaps not exactly what you might want

Why Is This Dangerous?

- Values from one function “leak” into another function
- If attacker can influence the values in the first function,
- Maybe he can alter the behavior of the second one

Variable Cleanup

- Often, programs reuse a buffer or other memory area
- If old data lives in this area, might not be properly cleaned up
- And then can be treated as something other than what it really was
- E.g., bug in Microsoft TCP/IP stack
  - Old packet data treated as a function pointer

Use-After-Free Bugs

- Increasingly popular security bug type
- Related to memory management
  - Memory structures are dynamically allocated on the heap
- Either explicitly or implicitly freed
  - Depending on language and context
- In some cases, pointers can be used to access freed memory
  - E.g., in C and C++

An Example Use-After-Free Bug

- In OpenSSL (from 2009)

```
. . .
frag->fragment,frag->msg_header.frag_len);
}
dtls1_hm_fragment_free(frag);
pitem_free(item);

if (al==0)
{
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

What Was the Effect?

- Typically, crashing the program
- But it would depend
- When combined with other vulnerabilities, could be worse
- E.g., arbitrary code execution
- Often making use of poor error handling code

Recent Examples of Use-After-Free Bugs

- Internet Explorer (2014, several in 2012-2013)
- Adobe Flash (2016, multiple cases in 2015)
- Mozilla, multiple products (2012)
- Google Chrome (2012)

Some Other Problem Areas

- Handling of data structures
  - Indexing error in DAEMON Tools
- Arithmetic issues
  - Integer overflow in Adobe Flash (2016)
  - Signedness error in XnView (2012)
- Errors in flow control
  - Samba error that causes loop to use wrong structure
- Off-by-one errors
  - Denial of service flaw in Clam AV (2011)

Yet More Problem Areas

- Null pointer dereferencing
  - FreeBSD denial of service (2016)
- Side effects
- Punctuation errors
- Typos and cut-and-paste errors
  - iOS vulnerability based on inadvertent duplication of a goto statement (2014)
- There are many others

Why Should You Care?

- A lot of this stuff is kind of exotic
- Might seem unlikely it can be exploited
- Sounds like it would be hard to exploit without source code access
- Many examples of these bugs probably unexploitable

So . . .?

- Well, that's what everyone thinks before they get screwed
- "Nobody will find this bug"
- "It's too hard to figure out how to exploit this bug"
- "It will get taken care of by someone else"
  – Code auditors
  – Testers
  – Firewalls

That's What They Always Say

- Before their system gets screwed
- Attackers can be very clever
  - Maybe more clever than you
- Attackers can work very hard
  - Maybe harder than you would
- Attackers may not have the goals you predict

But How to Balance Things?

- You only have a certain amount of time to design and build code
- Won't secure coding cut into that time?
- Maybe
- But less if you develop code coding practices
- If you avoid problematic things, you'll tend to code more securely

Some Good Coding Practices

- Validate input
- Be careful with failure conditions and return codes
- Avoid dangerous constructs
  - Like C input functions that don’t specify amount of data
- Keep it simple