Black Hat Webcast Series

C/C++ AppSec in 2014

LeafSR

Who Am I

- Chris Rohlf
- **Leaf SR** (Security Research) - Founder / Consultant
- BlackHat Speaker { 2009, 2011, 2012 }
- BlackHat Review Board Member
- **http://leafsr.com** **@chrisrohlf** **info@leafsr.com**

LeafSR

Agenda

- What is Application Security (AppSec) in C/C++?
- C/C++ Vulnerabilities: Prevention, Discovery, Remediation
- Memory corruption and runtime protections
- Conclusion

What is AppSec?

- AppSec is a general term that means preventing, detecting, and fixing security vulnerabilities in an application
- Secure design, code audits, penetration testing, fuzzing, threat modeling, and security patches are just a few activities that can fit under this umbrella
- AppSec is normally part of a well developed SDLC process

AppSec and C/C++

- C/C++ is still ubiquitous in enterprise and desktop systems alike
    - *Mobile Applications, Financial Systems, Databases, Browsers, Browser Plugins, Document Readers*
- The term AppSec is commonly associated with web apps, web frameworks and mobile development [1]
- AppSec frameworks and guidelines sometimes ignore the unique security issues applications developed in C/C++ must solve

[1] An informal poll showed the word AppSec is most associated with XSS, OWASP and "Vendor Lunches"

C/C++ Vulnerabilities

- C/C++ are a lot different than higher level languages
- There is very little language runtime available to the developer to fall back on when faced with complexity or an error condition
- Many unique security issues come from the underlying design principles that define these languages

C/C++ Vulnerabilities

- Static typing system (weakly enforced)
    - Type conversion, type truncation
- Pointers, pointers to pointers, pointers to arrays of pointers, pointers to lists of arrays of pointers to pointers… pointers
- Manual string copies, size calculations, concatenation, wide char
- Raw memory management
    - No runtime provided garbage collection

LeafSR

C/C++ Vulnerabilities

- Your system understands three basic primitives
  - Read, Write, Execute
- Memory corruption vulnerabilities give an attacker control or partial influence over these *RWX* primitives
  - stack overflows, heap overflows, integer overflows, type confusion, use after free, double free, uninitialized memory, integer truncation, out of bounds read/write, TOCTOU, race conditions

C/C++ Vulnerabilities

- Many of these bug classes are familiar to C/C++ developers
- Modern protections make many of them difficult to exploit by reducing predictability
- Attackers adapt to bug classes that are easier to exploit generically or have reusable techniques across different applications and systems

Type Confusion

- When a type identifier of a data structure becomes out of sync with that data the potential for a type confusion is there
    - Result: treat an object of type ***A*** as if it were of type ***B***
- Common in applications that exchange complex binary formats such as virtual machine byte code (Flash), objects or structures over a local IPC mechanism (Chrome)
- Structures with tagged unions are good places to start auditing

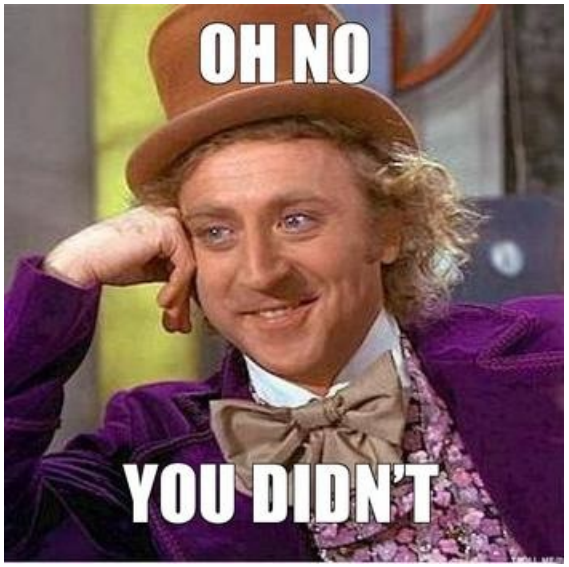
Type Confusion

- Type confusion in a C++ object with unsafe usage of the *reinterpret_cast* operator

```
class Widget {
  public:
    Widget() { }
    ~Widget() { }
    virtual void foo() { }
};

class Other {
  public:
    Other() { i = 0x41414141; }
    ~Other() { }
    int i;
};

void someFunc() {
    Other *o = new Other();
    Widget *b = reinterpret_cast<Widget *>(o);
    b->foo();
    delete o;
}
```



LeafSR

Use After Free

- Accessing an object after it has been deleted or free'd

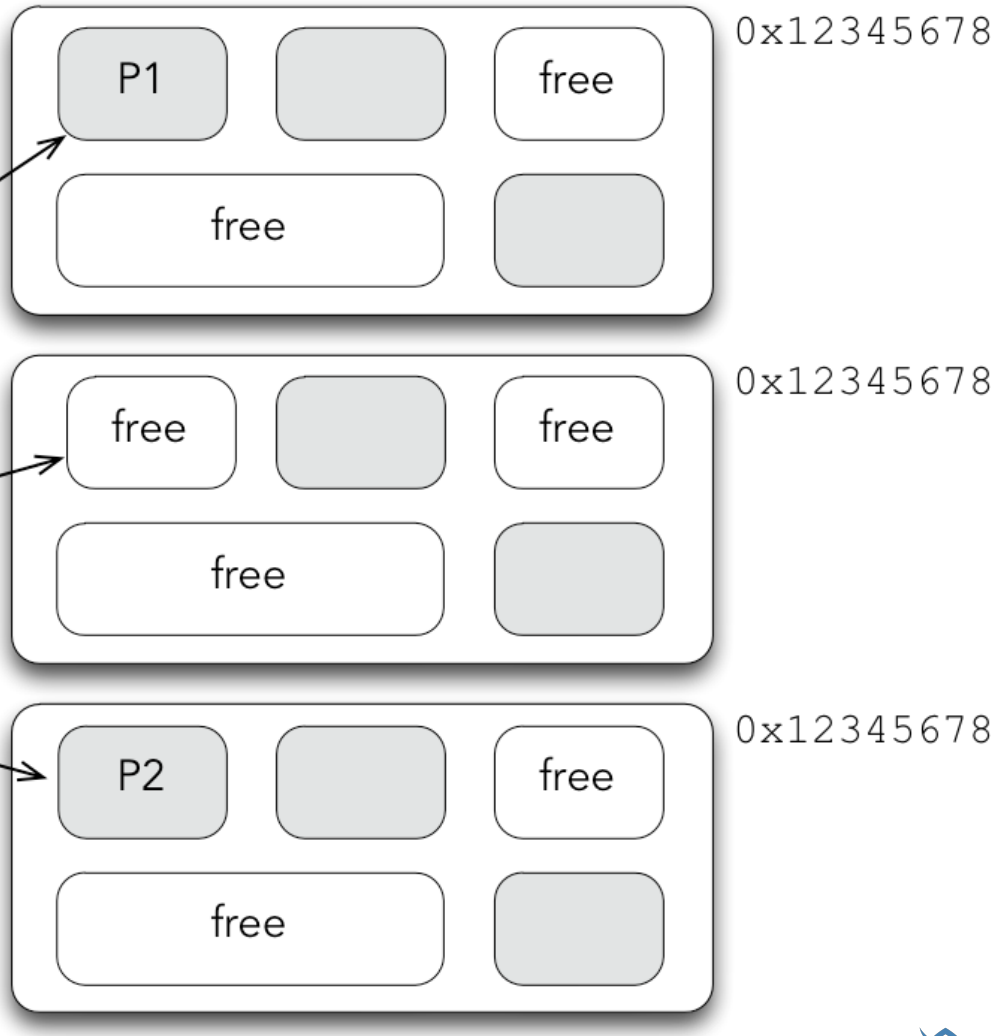
```
Buffers *someFunc(char *str) {
      char *a = (char *) malloc(1024);
      memcpy(a, str, 1023);
      doSomeStuff(a);
      free(a);
      doSomeOtherStuff(str);
      memcpy(m_bufs[0], a, 1023);
      return m_bufs;
}
```

- In C++ usually the result of poor object lifecycle management
  - Reference counting and garbage collection is one example

LeafSR

Use After Free (cont.)

```
P1 = new P1Obj();
...
delete P1;
...
P2 = new P2Obj();
...
P1->doSomething();
```



LeafSR

Use After Free (cont.)

- Complex applications contain many different components that must interact by exchanging objects of different types
- There must be a contract between these components that specifies a set of rules that will be followed for handling these objects safely
    - When these rules are violated we often see use-after-free patterns emerge
- Certain design patterns (e.g. JavaScript engine) make exploiting use-after-free vulnerabilities easier

Use After Free (cont.)

- Common use-after-free patterns include
  - Mixing smart pointers and raw pointers
  - Implementing a class without a matching copy constructor, assignment operator or destructor
  - Shallow copies that don't increment reference counts or copy whole objects

Vulnerability Prevention

- It is likely that your mobile app uses a closed source 3rd party library written in C
    - Keeping these up to date with relevant security patches is important
    - libpng, libjpeg, openssl are a few of examples
    - Yes, they will contain vulnerabilities too. But it is still better than writing your own version

Vulnerability Prevention

- Avoid common vulnerability patterns
  - Manual string concatenation
  - Mixing raw pointers and smart pointers
  - Allowing implicit conversions of signed/unsigned integers
  - Not defining hard limits on size and length values
    - Yes your protocol has a 32bit length member, do you really expect to transfer 4GB of data in a message?

Vulnerability Prevention

- Prevent the use of unsafe API calls
  - Microsofts *banned.h*
  - Custom GCC poison pragma - https://github.com/leafsr/gcc-poison

```
#pragma GCC poison strcpy


$ gcc -o string string.c


string.c: In function ‘main’:
string.c:8:2: error: attempt to use poisoned “strcpy”
```

LeafSR

Vulnerability Prevention

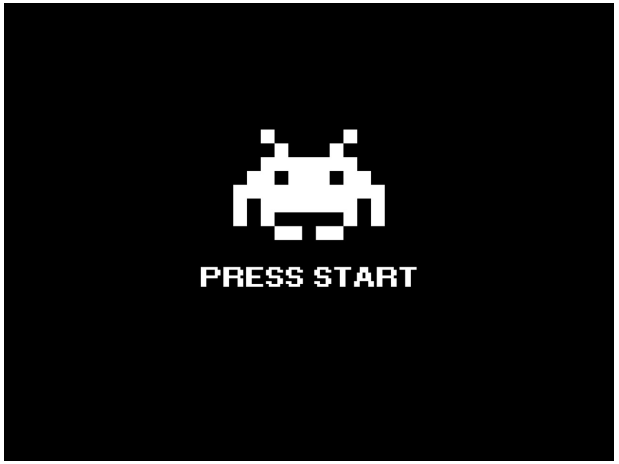
- Developer education
    - Study old vulnerabilities in your code reported by outside researchers or found by fuzzers
        - There is often a pattern to be extracted
- NIH? Don't reinvent the wheel, use an existing open source library if possible

Vulnerability Discovery

- Manual source code auditing with an IDE
- Time consuming and tedious but results in deeper and more subtle findings
- Start by looking at previously patched vulnerabilities in an application, identify the pattern, and find more instances like it
- Manual code audits give you a clearer root cause analysis of vulnerabilities in your applications, which allows you to better understand their severity

Vulnerability Discovery

- Sending malformed data to an application with the intent of monitoring for unexpected behavior such as an exception or a crash



- Fuzzing: cheap, fast, effective… shallow
  - Develop custom fuzzers or adapt open source ones to run against your code
    - Start by mutating existing unit-tests
  - Hardware is cheap, fuzz 24/7 against auto-generated builds of vour source tree

LeafSR

Vulnerability Discovery

- clang-analyzer
  - Source level analysis
  - Great for finding certain classes of bugs but requires your code be compiled with clang
- Address Sanitizer
  - Use in combination with fuzzing

LeafSR

Runtime Protections

- The issues faced by compiled C/C++ applications are very different than those in a web framework
- Exploiting memory corruption vulnerabilities on a modern operating system requires defeating memory protections
- Defeating these protections takes time and resources for an attacker, especially when they are combined

Runtime Protections

- ASLR - Address Space Layout Randomization
  - GCC: -fPIC -fPIE
  - Visual Studio: /DYNAMICBASE
- Ensures your process space is randomized at runtime
- This will reduce the reliability of exploits against your code that use deterministic properties of your application

Runtime Protections

- DEP - Data Execution Prevention
- Ensures that memory not marked executable cannot be executed
- Legacy systems may have to emulate this in software

Defensive Design

- The low level power and control of C/C++ gives us an opportunity to make exploit writers work for their money
- Study exploits for your application or one of a similar design
    - Reduce predictability and deterministic behavior
    - Examples: PartitionAlloc in Chrome, JIT hardening
- Sandboxes can help limit access by reducing privileges and separating resources from unprivileged components

Legacy Code

- Legacy code on a legacy system
  - If the application can be run in a sandbox this is likely to result in the best security ROI
  - Audit code for older / pre-SDLC bugs (`strcpy`, `sprintf`, `gets` and so on)
  - Fuzz, patch, fuzz, patch

LeafSR

Legacy Code

(I think these are facebooks servers)

- Legacy code on a modern system
  - Even older code benefits from operating system supplied protection for free, but this may require compatibility testing
  - Use a newer compiler to benefit from compiler added protections (stack cookies, SafeSEH, SEHOP)
    - MSVC 2010 or newer

Conclusion

- Audit, Fuzz, Audit, Fuzz, Audit, Fuzz …
- Enable any memory protections made available by the operating system for free, investigate which compiler protections you aren't currently utilizing in your code
- Stay up to date with attacker trends to help prioritize your efforts
- Study existing exploits and harden your application as necessary to reduce deterministic behavior

chris.rohlf @ leafsr.com

LeafSR