SoK: Eternal War in Memory
Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song
In: Oakland ‘14

Presenter: Mathias Payer, EPFL
http://hexhive.github.io
FFmpeg and a thousand fixes

>1,000 bugs found and fixed

2 person-years & fuzzing on large cluster
Software is unsafe and insecure

- Low-level languages (C/C++) trade type safety and memory safety for performance
  - Programmer responsible for all checks

- Large set of legacy and new applications written in C / C++ prone to memory bugs

- Too many bugs to find and fix manually
  - Protect integrity through safe runtime system
A Model for Memory Corruption
Memory (un-)safety: invalid deref.

Dangling pointer: (temporal)

Out-of-bounds pointer: (spatial)

Violation iff: pointer is read, written, or freed

```c
char foo[40];
foo[42] = 23;
free(foo);
*foo = 23;
```
Type Confusion

class B {
    int b;
};
class D: B {
    int c;
    virtual void d() {}
};
...
B *Bptr = new B;
D *Dptr = static_cast<D*>(B);
Dptr->c = 0x43; // Type confusion!
Dptr->d(); // Type confusion!
Attack scenario: code reuse

- Find addresses of gadgets
- Force memory corruption to set up attack
- Leverage gadgets for code-reuse attack
- (Fall back to code injection)
Benign control-flow

```c
void vuln(char *u1) {
    // strlen(u1) < MAX ?
    char tmp[MAX];
    strcpy(tmp, u1);
    ...
}
vuln(&exploit);
```
Control-flow hijack attack

```c
void vuln(char *u1) {
    // strlen(u1) < MAX ?
    char tmp[MAX];
    strcpy(tmp, u1);
    ...
}
vuln(&exploit);
```

- **Memory safety Violation**
- **Integrity**
- **Location**
- **Usage**
- **Attack**

- Don't care
- Don't care
- Points to &system()
- Base pointer after system()
- 1st argument to system()
Model for memory attacks

- Memory safety
- Integrity
- Location
- Usage
- Attack

Memory corruption

C
*C
D
*D

&C
&D

&&C
&&D

Code corruption
Control-flow hijack
Data-only
Data execution prevention

Memory safety

Integrity

Location

Usage

Attack

- Memory corruption
- Code corruption
- Control-flow hijack
- Data-only
Stack canaries and SEH

Memory safety

- Memory corruption

Integrity

- C
- *C
- D
- *D

Location

- &C

Usage

- &D

Attack

- Code corruption
- Control-flow hijack
- Data-only
Address space layout random.

Memory safety

Memory corruption

Integrity

C

*C

D

*D

Location

&C

&D

Usage

*C

&D

Attack

&&C

*&&D

Code corruption

Control-flow hijack

Data-only
ASLR: Performance overhead

- ASLR uses one register for PIC / ASLR code
  - Performance degradation on x86
Widely deployed defenses

Deployed defenses incomplete and not effective

- Memory
- Integrity
- Location
- Usage
- Attack

- Code corruption
- Control-flow hijack
- Data-only
Defense strategies

Stop memory corruption

- Safe dialects of C/C++: CCured, Cyclone
- Retrofit on C/C++: SoftBounds+CETS
- Rewrite in safe language: Java/C#

**Memory safety** → **Violation**

**Integrity**

**Randomization**

**Flow Integrity**

**Attack** Control-flow hijack
Defense strategies

Enforce integrity of reads/writes
- Write Integrity Testing
- (DEP and W^X for code)
Defense strategies

Probabilistic defenses

- Randomize locations, code, data, or pointer values
Defense strategies

Protect control transfers
- Data-flow integrity
- Control-flow integrity
Model for memory attacks

- Model allows reasoning and classification
  - Classify security policies and defense mechanisms
  - Reason about power of attacks

- Identify properties that enable wide adoption
  - Low overhead is key (<10%)
  - Compatibility with legacy code and source code
  - Protection against class(es) of attacks
Conclusion
Conclusion

• Low level languages are here to stay
  – We need protection against memory vulnerabilities
  – Enforce performance, protection, compatibility

• Mitigate control-flow hijack attacks
  – Secure execution platform for legacy code
  – Code-pointer integrity for source code

• Future directions: strong policies for data
  – Protect from other attack vectors