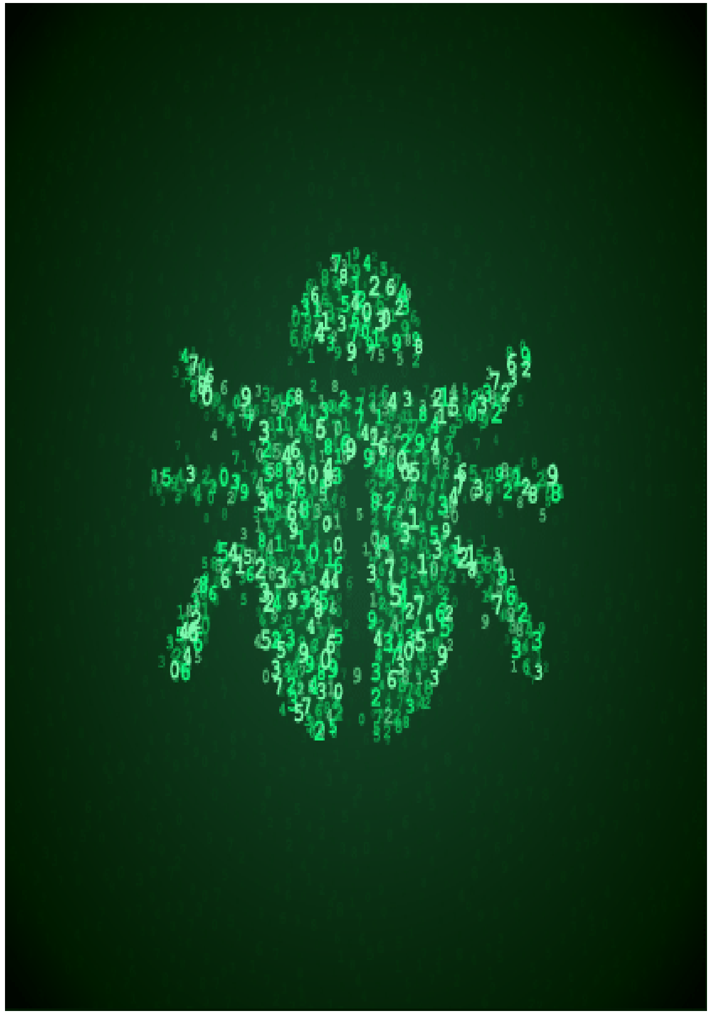
Automated Code Repair

Will Klieber

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0004068

Automated Code Repair – Motivation



Software vulnerabilities constitute a major threat

- A majority arise from common coding errors
- Shown by experience from source code analysis labs at CERT and DoD

Static analysis tools help, but:

- Typically are used late in the development process
- Produce an enormous number of warnings
- The volume of true positives often overwhelms the ability of the development team to fix the code

Huge amount of code in use by DoD

- Billions of lines of C code
- Unknown number of security vulnerabilities

Integer Overflow

This past year (FY16), we developed techniques for automated repair of **integer overflows** that lead to **memory corruption**

Integers in C are represented by a fixed number of bits *N* (e.g., 32 or 64).

- Overflow occurs when the result cannot fit in *N* bits
- Modular arithmetic: Only the least significant *N* bits are kept

How does integer overflow lead to memory corruption?

1. Memory allocation: `malloc(·)`.
2. Bounds checks for an array

Example: Android Stagefright bugs (July 2015)

Benefits to DoD

Eliminate security vulnerabilities at a **much lower cost** than manual repair

Integer overflows are a **very common** type of bug

- In CERT SCALe audits, about 80% of findings were related to fixed-width integers

Our technique:

- **Will not break working code**, provided *inferred specification* is correct (Next slide)
- Typically total slowdown < 5% (Based on theoretical model)
- False positives: Flagged operations that cannot actually overflow
  - Then our ‘repair’ just adds a little unnecessary overhead

Premises for Automated Repair

1. Many security bugs follow common patterns
   - E.g., "`p = malloc(n * sizeof(T))`" where *n* is attacker-controlled
   - Integer overflow $\Longrightarrow$ too little memory gets allocated
2. By recognizing such a pattern, it is possible to make a reasonable guess of the developer's intention (the ***inferred specification***)
   - E.g., "Try to allocate enough memory to hold *n* objects of type *T*"
3. It is possible to repair the code to satisfy this inferred specification
   - Check if overflow occurs; if so, simulate `malloc` failing with `ENOMEM`

Experimental Results

	Overflows (as reported by Kint)	Overflows that are *sensitive*	Overflows fully repaired	Semi-repair	Unrepaired
OpenSSL (1.0.2g)	969	233	180	28	25
Jasper	481	101	53	32	16

An overflow is ***sensitive*** if it involves variables that are associated with array indices or bounds

Note: Some of the above “repairs” are actually false positives (i.e., operation never overflows). Others are known vulnerabilities with CVEs and patches.

Repair Strategy

Inferred specification: **inequality comparisons** involving array indices or bounds should behave as if normal arithmetic (not modular arithmetic) were used

- Includes `malloc`
- Excludes crypto and hashing functions

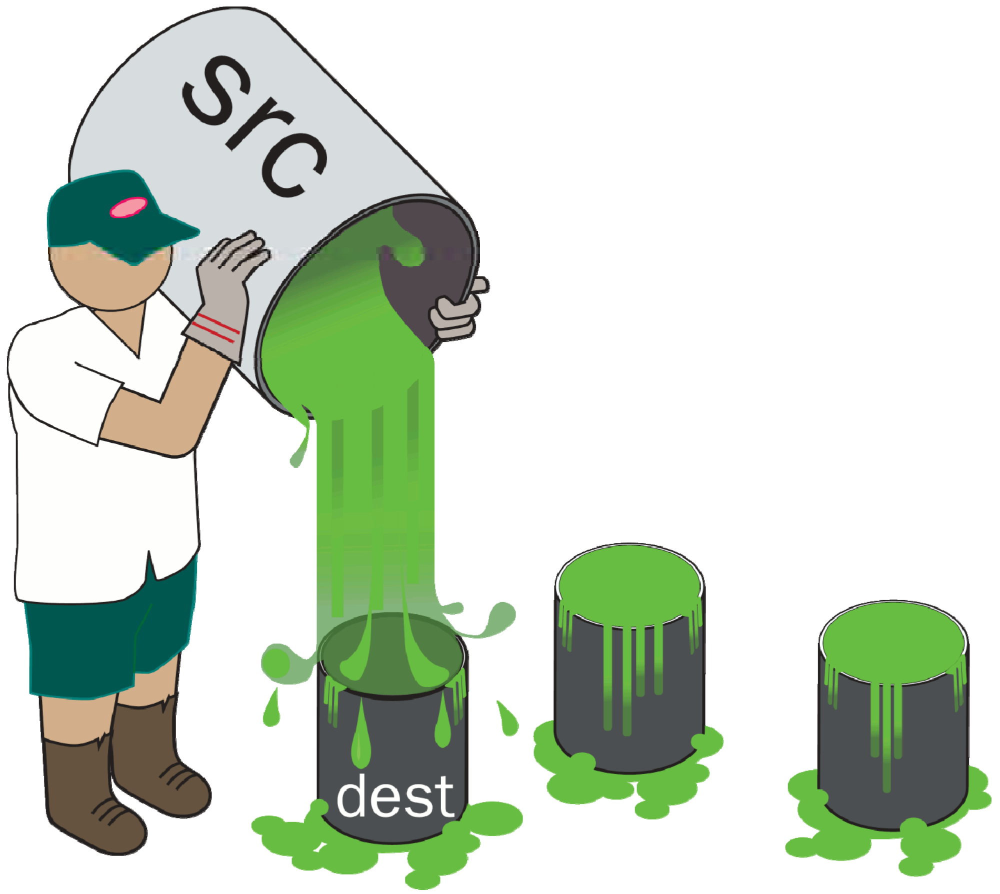
Repair: General case is intractable (with bounded memory)

Special case that we handle: non-negative integers with only addition or multiplication (no subtraction or division)

- The value is monotonically non-decreasing (except for multiplication by zero)
- Normal arithmetic can be emulated using **saturation arithmetic**:
  - Replace an overflowed value with the greatest representable value (`SIZE_MAX`)
- If the declared types of variables are smaller than `size_t`, they are promoted up

Arithmetic for Checking Bounds of an Array

Example: copy *n* bytes from *src* to *dest*, starting at index *start* of *dest*.



Repair: **UADD(start, n)** /* defined on next slide */

```
if (start + n <= dest_size) {
  memcpy(&dest[start], src, n);
} else {
  return -EINVAL;
}
```

wrappers.h

```
inline static size_t UADD(size_t lop, size_t rop) {
    size_t result;
    bool flag = __builtin_add_overflow(lop, rop, &result);
    if (flag) {result = SIZE_MAX;}
    return result;
}
```

Repair: **UADD(start, n)**

```
if (start + n <= dest_size) {
  memcpy(&dest[start], src, n);
} else {
  return -EINVAL;
}
```

- What if `dest_size` is `SIZE_MAX`?
- What if both sides of inequality overflow?
- What if overflow reaches a non-comparison sink?

Semi-Repair

Example adapted from CVE-2015-8370:

```
unsigned cur_len = 0;
while (1) {
  key = grub_getkey();
  if (key == '\b') {
    if (cur_len == 0) {
      /* FIXME: Insert error-
         handling code here. */
    }
    cur_len--;
    grub_printf("\b");
    continue;
  }
  if (cur_len + 2 < buf_size) {
    buf[cur_len++] = key;
    grub_printf("%c", key);
  }
}
```

semi-repair

If a potentially overflowed value is used to index into an array, do a semi-repair.

Tool inserts check for overflow.
User writes error-handling code.

Future Directions

In FY17, we have two Automated Repair projects:

1. **Inference of memory bounds**
   - Buffer overflow (WRITEs) and leakage of sensitive information (READs)
2. **Incorrect usage of crypto/security APIs**
   - E.g., incorrect validation of certificate chain using OpenSSL API, leading to MITM

A difficulty we encountered was the **Source↔IR mapping problem:**

- Code is most readily analyzed and repaired on an intermediate representation (IR)
- Transformations on the IR aren't unambiguously mappable to the source
- Macros and `#ifdefs` are a further difficulty
  - Prof. Christian Kästner (CMU SCS) has done work on `#ifdefs` as part of this project
- We are further investigating these issues this year (FY17)

Conclusion

Automated code repair (ACR) reduces a system’s attack surface and improves its ability to withstand cyber-attacks.

ACR is suitable for problems where many bugs follow a common pattern and have a corresponding pattern for repair.

In FY16, we focused on integer overflows involving memory bounds/indices.

We are continuing work on ACR in FY17.

Contact Information

Will Klieber

Software Security Researcher

Telephone: +1 412.268.9207

Email: weklieber@cert.org