Automated Code Repair

Integer overflow in calculations related to array bounds or indices is almost always a bug. We have developed and implemented an automated technique for repairing such bugs so that the program behaves as likely desired.

Experience from source code analysis labs at CERT and DoD shows that most software contains numerous vulnerabilities. A majority arise from common coding errors.

Static analysis tools help, but typically they produce an enormous number of warnings. The volume of just the true positives can overwhelm the ability of the development team to fix the code. Consequently, the team eliminates only a small percentage of the vulnerabilities.

Our work on automated repair is based on three premises

1. **Many security bugs follow common patterns.** E.g., one common bug pattern is "`p = malloc(n * sizeof(T))`" where ***n*** is attacker-controlled. If ***n*** is very large, integer overflow occurs, and too little memory is allocated. This sets the stage for a buffer overflow later on.
2. **By recognizing such a pattern, it is possible to make a reasonable guess of the developer's intention (inferred specification).** E.g., "Try to allocate enough memory for ***n*** objects of type T."
3. **It is possible to repair the code to satisfy this inferred specification.** Example of repair: Insert code to check if overflow occurs and, if it does, to simulate `malloc` failing with ENOMEM.

Example:
copy ***n*** bytes
from src to dest ,
starting at index ***start***
of dest, and ending at
index `start`+***n*−1.**

Integer Overflow

Integers in C are stored in a fixed number of bits ***N*** (e.g., 32 or 64). Overflow occurs when the result cannot fit in ***N*** bits.

In modular arithmetic, only the least significant ***N*** bits are kept.

This past year (FY16), we focused on integer overflow that leads to memory corruption. E.g.:

- Memory allocation: malloc(n), where the calculation of n can overflow.
- Integer overflow in array bounds check.

Example: Android Stagefright vul (July 2015) had both of the above types of overflows.

Repair: Emulate normal arithmetic

For non-negative integers with only addition or multiplication (no subtraction or division), the value is **monotonically non-decreasing** (except for multiplication by zero).

In this case, unlimited-bitwidth arithmetic can be emulated by using saturation arithmetic: Replace an overflowed value with the greatest representable value.

wrapper.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;
}
```

If a potentially overflowed value is used to index into an array, do a semi-repair (add a check to detect overflow, ask user to write error-handling code).

Example semi-repair from CVE-2015-8370

```
1.  unsigned cur_len = 0;
2.  while(1) {
3.  key = grub_getkey();
4.     if (key == '\b') {
5.        if (cur_len == 0) {
6.           /* Add error-handling
             code here. */
7.        }
8.        cur_len--;
9.        grub_printf("\b");
10.       continue;
11.    }
12.    if (cur_len + 2 < buf_size) {
13.       buf[cur_len++] = key;
14.       grub_printf("%c", key);
15.    }
16. }
```

Experimental Results

	OpenSSL	Jasper
Overflows*	969	481
Overflows that are sensitive	233	101
Overflows fully repaired	180	53
Semi-repair	28	32
Unre-paired	25	16

*(as reported by Kint)

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

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 security bugs follow a common pattern and have a corresponding pattern for repair.

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

A difficulty we encountered was the Source<->IR mapping problem

- Code is most readily analyzed and repaired on an intermediate representation (IR). But actual repair must be on the source.
- Transformations on the IR aren't unambiguously mappable to the source.
- Macros and #ifdefs are a further difficulty.
- We are continuing to investigate these issues in FY17.

Contact: Will Klieber | weklieber@sei.cmu.edu