Retrofitting Security in input parsing routines

Jayakrishna Menon, Christophe Hauser, Yan Shoshitaishvili, Stephen Schwab
{jmenon, hauser, schwab}@isi.edu
yans@asu.edu
Modern defenses

- OS defenses (ASLR, DEP).
- Compiler-level defenses (e.g., stack canaries).
- Code audit tools.

Vulnerabilities

- Many programs are still written in unsafe languages like C/C++.
- Memory corruption vulnerabilities remain prominent.
PARSERS

- Directly exposed to user input.
- Many custom implementations in unsafe languages (C/C++).
- Over 170 vulnerabilities reported in various parsing mechanisms since 1999.
- Varying semantics and the abundance of string manipulations make their implementation error-prone.
Solution space
Design Time Security

- Parser libraries.
- Parser generators.
- Formal methods.

Post-Design Security

- Code audits.
- Refactoring/inserting correct parsers.
- No source code?
**Binary-level approach**

- Source code not always available (legacy code, uncooperative editors, untrusted IoT devices).
- What you see is not what you execute: compiler bugs, compiler “backdoors” e.g., XCodeGhost (linking malicious code into executables).

WYSINWYX
CHALLENGES
Scaling problem

Program analysis techniques are difficult to automate in a scalable and precise manner.
Static analysis

- Scalable.
- Imprecise.

Symbolic execution

- Precise.
- Unscalable.
Dynamic analysis

- Precise.
- Low coverage.
Source code

- Types.
- Variable names.
- Functions.
- ...

Binary

- Registers.
- Memory locations.
- Basic blocks.
- ...

...
How to scale to real world programs?
TEMPLATE-BASED APPROACH

... to discover vulnerabilities based on templates corresponding to common classes of security bugs.

... to retrofit security by patching programs at the binary-level.
**Initial Approach**

- Focuses on overflows in buffers allocated statically on the stack.
- Template-based: categorize causes of vulnerabilities into three classes.
- Combines static analysis and symbolic execution.

**Classes/Templates**

- Unconstrained input.
- Under-constrained input size.
- Unchecked termination condition.
- ...
Unconstrained input.

Improper usage of functions that do not check for sizes such as strcpy, sprintf etc.
Example 1: CVE-2003-0390

```c
int opt_atoi( char *s) {
    char buf[1024];
    char *fmt = "String [%s] is not valid";
    sprintf(buf, fmt, s);
}
```
Under-constrained input size.

Improper validation of size field in functions such as memcpy.
Example 2: CVE-2015-3329

```c
void phar_set_inode( phar_entry_info *entry) {
    char tmp[1024];
    memcpy(tmp, entry->phar->fname, entry->phar->fname_len);
}
```
Unchecked termination condition.

Performing operations on (possibly) incorrectly terminated strings.
2-step Analysis approach

Static analysis

- CFG
  - Identify string manipulation functions.
- DDG
  - Identify user input.
  - Analyze backward data-dependency.

Symbolic analysis

- SE
  - Dangerous program paths.
  - Path constraints.

(Memory corruption caused by unsafe buffer manipulation)
## Analysis results

<table>
<thead>
<tr>
<th></th>
<th>Static Analysis</th>
<th>Symbolic execution</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>False positive rate</strong></td>
<td>6.6%</td>
<td>0%</td>
<td>0% *</td>
</tr>
<tr>
<td><strong>False negative rate</strong></td>
<td>40%</td>
<td>0% *</td>
<td>40%</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>1-260s</td>
<td>1-400s</td>
<td>2-660s</td>
</tr>
</tbody>
</table>
2 new bugs found in the binary code of common opensource projects and libraries (in a semi-automatic setting)
Retrofitting security: binary patching
Adding the missing checks

- Remember: we focus on stack buffers.
- On the identified program paths, we constrain the user input such that:
  
  \[
  \text{user\_input\_size} < \text{stack\_buffer\_size}
  \]
Adding the missing checks

When the constraints are violated, we crash the program. This is equivalent to e.g., _sprintf_chk()
Patching the binary

Static reassembly problems: breaking internal program references.

Partial solution: inject trampoline gadgets in padding bytes between functions (up to 15 consecutive NOPs).
# Inserting Checks

```c
int opt_atoi(char *s)
{
    sprintf(buf, fmt, s);
}
```

```c
int opt_atoi(char *s)
{
    if(strlen(s)>1024)
    {
        exit();
        sprintf(buf, fmt, s);
    }
}
```
More templates
New template

Memory allocation errors

... authentication errors.

... misuses of cryptographic APIs.

... information leakage.
New bugs

12 new bugs found in the binary code of common open source programs and libraries (in a fully automated setting).
DISCUSSION

Lightweight and scalable approach.

... but high rate of false negatives.

... limited patching capabilities.
Stumbling blocks

Data structure recovery.

Pointer aliasing.
Future work

- Improve data dependence tracking.
- Leverage static reassembly techniques.
- More vulnerability templates.
- Apply to large corpus of IoT firmware.
Key takeaways

- Templates per vulnerability class.
- Scalable, two-level approach based on a combination of static analysis + symbolic execution.
- High-precision: we can infer semantic-agnostic patches for each class.
- New bugs.