Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison

Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu
Why do we need to identify security bugs?
Motivation

- The overwhelming number of bugs reports
  - Mozilla: ~ 300 bugs reports per day
  - Linux kernel: More than 900K commits have been made
    - ~165 git commits per day
  ...
Motivation

- The overwhelming number of bugs reports
- Patch propagation in derivative programs is hard and expensive
  - Example: Many projects are derived from the Linux kernel
Motivation

- The overwhelming number of bugs reports
  - Security bugs may not be fixed timely, and attackers have opportunities to exploit these security bugs

- Patch propagation in derivative programs is hard and expensive

Maintainers are prioritizing to fix security bugs. Unrecognized security bugs may be left unpatched!
Our goal:

Identify patches that are for security bugs
How to identify patches for security bugs?
Traditional approaches:

- **Text-mining**
  - Analyze textual information of patches to find security-related keywords.

- **Statistical analysis**
  - Differentiate patches of security bugs from general bugs by using statistical information.

Limitations:

1. Bad precision.
2. Cannot know the security impacts of bugs.
Limitations of traditional approaches:

CVE-2014-8133 Permission bypass

commit 41bdc78544b8a93a9c6814b8bbbfef966272abbe
Author: Andy Lutomirski <luto@amacapital.net>
Date:   Thu Dec 4 16:48:16 2014 -0800

x86/tls: Validate TLS entries to protect espfix

Installing a 16-bit RW data segment into the GDT defeats espfix. AFAICT this will not affect glibc, Wine, or dosemu at all.

Signed-off-by: Andy Lutomirski <luto@amacapital.net>
Acked-by: H. Peter Anvin <hpa@zytor.com>
Cc: stable@vger.kernel.org
Cc: Konrad Rzeszutek Wilk <konrad.wilk@oracle.com>
Cc: Linus Torvalds <torvalds@linux-foundation.org>
Cc: security@kernel.org <security@kernel.org>
We prefer a program analysis-based method

- Understand the semantics of patches and bugs precisely
- A bug is a security bug if it causes *security impacts* when triggered.
- A patch is for a security bug when it blocks the security impacts
How to know if a patch blocks security impacts?
A security impact = A security-rule violation

Security rules are coding guidelines used to prevent security bugs.

Security-rule violations cause security impacts.
We thus check if a patch blocks security-rule violations.
Common security rules

Rule 1: In-bound access
Read & write operations should be within the boundary of the current object.

Rule 2: No use after free
An object pointer should not be used after the object has been freed.

Rule 3: Use after initialization
A variable should not be used until it has been initialized.

Rule 4: Permission check before sensitive operations
Permissions should be checked before performing sensitive operations, such as I/O operations.
Violations for common security rules

Rule 1: In-bound access
   ↓ violation
Out-of-bound access

Rule 2: No use after free
   ↓ violation
Use-after-free

Rule 3: Use after initialization
   ↓ violation
Uninitialized use

Rule 4: Permission check before sensitive operations
   ↓ violation
Permission bypass
A patch blocks security impacts if:

If we can prove the following conditions:

**Condition 1:** The unpatched version of code violates a security rule.

**Condition 2:** The patched version of code does not violate the security rule.
Challenge:

How to precisely determine the security-rule violations?
Intuition:

We can leverage two unique properties of under-constrained symbolic execution.
Property 1: Constraints model violations

Security-rule violations can be modeled as constraints

Example:

Buffer access: Buffer[Index];

Constraints for out-of-bound access:

\[ Index \geq UpBound, \text{ and/or } Index \leq LowBound \]
Property 2: Conservativeness

Under-constrained symbolic execution is conservative.

- False-positive solutions
  - If the constraints are solvable, this can be a false positive.

- Proved unsolvability
  - If it cannot find a solution against constraints, they are indeed unsolvable.
Leverage the properties for determining the security-rule violations

- Patch-related operations can be modeled as symbolic constraints
- To show the patched version won’t violate a security rule
  - To prove “violating” is unsolvable
- To show the unpatched version will violate the security rule
  - To prove “non-violating” is unsolvable
Our approach: Symbolic rule comparison

1. Construct **opposite** constraint sets for the patched and unpatched version
   a. Patched version: Construct constraints for violating security rules
   b. Unpatched version: Construct constraints for not violating security rules
2. Check the **unsolvability** of these constraint sets
3. Confirm the patches for security bugs if both constraint sets are unsolvable
Rationale behind our approach

- For a security rule, the patched version NEVER violate it
  - This means that the patched version is in a safe state
Rationale behind our approach

- For a security rule, the patched version NEVER violate it
  - This means that the patched version is in a safe state

- In the situations that opposite to conditions of the patch, the unpatched version MUST violate this security rule
  - This means that the unpatched version is in an unsafe state
Rationale behind our approach

- For a security rule, the patched version NEVER violate it
  - This means that the patched version is in a safe state

- In the situations that opposite to conditions of the patch, the unpatched version MUST violate this security rule
  - This means that the unpatched version is in an unsafe state

- The patch changes the code from an unsafe state to a safe state
  - Precisely confirmed with property 2
Rationale behind our approach

- For a security rule, the patched version NEVER violate it
  - This means that the patched version is in a safe state

- In the situations that opposite to conditions of the patch, the unpatched version MUST violate this security rule
  - This means that the unpatched version is in an unsafe state

- The patch changes the code from an unsafe state to a safe state

The patch fixed a security bug with the security impact that corresponding to the security rule violation.
A concrete example
STEP 1: Symbolically analyzing patched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(... , u8 sta_id) {
    if (sta_id >= IWLAGN_STATION_COUNT) {
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
        return -EINVAL;
    }
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u 
                  "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);
    ...
    return 0;
}
```
STEP 1: Symbolically analyzing patched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(... , u8 sta_id) {
  if (sta_id >= IWLAGN_STATION_COUNT) {
    IWL_ERR(priv, "invalid sta_id %u", sta_id);
    return -EINVAL;
  }
  if (!(priv->stations[sta_id].used ))
    IWL_ERR(priv,"Error active station id %u "
             "addr %pM\n",
             sta_id, priv->stations[sta_id].sta.sta.addr);
  ...
  return 0;
}
```

Identify security operations.
STEP 1: Symbolically analyzing patched code

```
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
  if (sta_id >= IWLAGN_STATION_COUNT) {
    IWL_ERR(priv, "invalid sta_id %u", sta_id);
    return -EINVAL;
  }
  if (!(priv->stations[sta_id].used ))
    IWL_ERR(priv,"Error active station id %u "
             "addr %pM\n",
             sta_id, priv->stations[sta_id].sta.sta.addr);
  ...
  return 0;
}
```
STEP 1: Symbolically analyzing patched code

Identify security operations.

Extract critical variable.

Identify vulnerable operations.

Slicing

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (sta_id >= IWLAGN_STATION_COUNT) {
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
        return -EINVAL;
    }

    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
                 "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);

    ... return 0;
}
```
STEP 2: Collecting and construct constraints for patched code

```c
// CVE-2012-6712
int iwl_stas_ucode_activate(..., u8 sta_id) {
    if (sta_id >= IWLAGN_STATION_COUNT) {
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
        return -EINVAL;
    }
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
            "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);
    ... return 0;
}
```

### Collecting constraints

<table>
<thead>
<tr>
<th>Constraints source</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security operations</td>
<td>sta_id &lt; IWLAGN_STATION_COUNT</td>
</tr>
<tr>
<td>Slice</td>
<td>N/A</td>
</tr>
<tr>
<td>Artificial constraints (Security rules)</td>
<td>sta_id &gt;= Bound of priv-&gt;stations</td>
</tr>
</tbody>
</table>
STEP 3: Solving constraints for patched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (sta_id >= IWLAGN_STATION_COUNT) {
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
        return -EINVAL;
    }

    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
                "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);

    ... return 0;
}
```

Collecting constraints

<table>
<thead>
<tr>
<th>Constraints source</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security operations</td>
<td>sta_id &lt; IWLAGN_STATION_COUNT</td>
</tr>
<tr>
<td>Slice</td>
<td>N/A</td>
</tr>
<tr>
<td>Artificial constraints (Security rules)</td>
<td>sta_id &gt;= Bound of priv-&gt;stations</td>
</tr>
</tbody>
</table>

These constraints are unsolvable!
STEP 3: Solving constraints for patched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(... , u8 sta_id) {
    if (sta_id >= IWLAGN_STATION_COUNT) {
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
        return -EINVAL;
    }

    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u 
            "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);

    ... return 0;
}
```

The patched version **won’t** violate the security rule.

These constraints are unsolvable!
STEP 1’: Symbolically analyzing unpatched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u 
                 "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);
    ... return 0;
}
```

Identify vulnerable operations.
STEP 1’: Symbolically analyzing unpatched code

```
// CVE-2012-6712
int iwl_sta_ucode_activate(... , u8 sta_id) {
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u ",
                 "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);
    ...
    return 0;
}
```

Extract critical variable.

Identify vulnerable operations.
STEP 1’: Symbolically analyzing unpatched code

```c
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
                 "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);

    ... return 0;
}
```

- Extract critical variable.
- Identify vulnerable operations.

Slicing
STEP 2’: Collecting and construct constraints for unpatched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (!(priv->stations[sta_id].used))
        IWL_Err(priv,"Error active station id %u "
             "addr %pM\n",
             sta_id, priv->stations[sta_id].sta.sta.addr);
    ...
    return 0;
}
```

Collecting constraints

<table>
<thead>
<tr>
<th>Constraints source</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security operations</td>
<td>-</td>
</tr>
<tr>
<td>Slice</td>
<td>-</td>
</tr>
<tr>
<td>Artificial constraints (Security rules)</td>
<td>sta_id &lt; Bound of priv-&gt;stations</td>
</tr>
</tbody>
</table>
STEP 2': Collecting and construct constraints for unpatched code

Collecting constraints

<table>
<thead>
<tr>
<th>Constraints source</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security operations</td>
<td>sta_id &gt;= IWLAGN_STATION_COUNT</td>
</tr>
<tr>
<td>Slice</td>
<td>-</td>
</tr>
<tr>
<td>Artificial constraints</td>
<td>sta_id &lt; Bound of priv-&gt;stations</td>
</tr>
</tbody>
</table>

```
1 // CVE-2012-6712
2 int iwl_stu_ucode_activate(..., u8 sta_id) {
3     if (!(priv->stations[sta_id].used ))
4         IWL_ERR(priv,"Error active station id %u "
5             "addr %pM\n",
6             sta_id, priv->stations[sta_id].sta.sta.addr);
7     ...
8     return 0;
9 }
```
STEP 3': Solving constraints for unpatched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {
    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
                 "addr %pM\n", sta_id, priv->stations[sta_id].sta.sta.addr);

    return 0;
}
```

Slicing & Collecting constraints

<table>
<thead>
<tr>
<th>Constraints source</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security operations</td>
<td>sta_id &gt;= IWLAGN_STATION_COUNT</td>
</tr>
<tr>
<td>Slice</td>
<td>-</td>
</tr>
<tr>
<td>Artificial constraints (Security rules)</td>
<td>sta_id &lt; Bound of priv-&gt;stations</td>
</tr>
</tbody>
</table>

These constraints are also unsolvable!
STEP 3’: Solving constraints for unpatched code

```c
// CVE-2012-6712
int iwl_sta_ucode_activate(..., u8 sta_id) {

    if (!(priv->stations[sta_id].used ))
        IWL_ERR(priv,"Error active station id %u "
                 "addr %pM\n",
                 sta_id, priv->stations[sta_id].sta.sta.addr);

    ...
    return 0;
}
```

The unpatched version MUST violate the security rule.

These constraints are also unsolvable!
STEP 4: Symbolic rules comparison

- The constraints for patched version are unsolvable!
  - “Violating security rules” is unsolvable
  - Patched version does not have an out-of-bound access

- The constraints for unpatched version are unsolvable!
  - “NOT violating security rules” is unsolvable
  - Unpatched version has out-of-bound accesses

Conclusion: The patch blocks an out-of-bound access.
Advantages of our approach

- Very few false positives --- Special use of under-constrained symbolic execution
  - 97% precision rate

- Determine security impacts of bugs
  - By detecting security rules violations, it can identify security bugs and also their security impacts

- Easy to extend
  - To cover more kinds of security impacts, users just need to model more types of security rules
Implementation

- Our prototype: SID
  - Based on LLVM

- Currently support five types of common security impacts
  - Out-of-bound access, permission bypass, uninitialized use, use-after-free, and double-free
Evaluation
Performance

- We analyzed 54K patches
- The experiments were performed on a desktop with 32GB RAM and 6 core Intel Xeon CPU
- The analysis takes an average of 0.83 seconds for each patch.
False-positive and false-negative analysis

- Few false positives
  - We confirmed 227 security bugs with 8 false-positive cases.

- False negatives (can be reduced)
  - 53% false negatives.
  - Most of them are caused by incomplete coverage for security and vulnerable operations.
Security evaluation for identified security bugs

- **Security impacts**
  - Already confirmed by SID

- **Reachability**
  - Check the call chain from entry points to vulnerable functions
Security evaluation for identified security bugs

● Vulnerability confirmation for CVE
  ○ 54 CVEs confirmed out of 227 identified bugs.
  ○ 117 security bugs are still under review.

● Reachability analysis for security bugs
  ○ 28 dynamically confirmed bugs (fuzzers).
  ○ 154 are reachable from attacker controllable entry points, such as system calls.

● 21 security bugs still unpatched in the Android kernel.
Conclusions

● Timely patching of security bugs requires the determination of security impacts
  ○ Patch propagation is hard and expensive
  ○ So maintainers have to prioritize to fix the security bugs.

● We exploit the properties of under-constrained symbolic execution for the determination
  ○ Our novel approach: Symbolic rule comparison

● Identified many overlooked security bugs in the kernel
  ○ They may cause critical security consequences
<table>
<thead>
<tr>
<th>Main security impacts</th>
<th>Security rules violation</th>
<th>Common fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bound access (16.5%)</td>
<td>Read/Write out of boundary</td>
<td>Add bound check (79%)</td>
</tr>
<tr>
<td>Uninitialized use (13.7%)</td>
<td>Use before initialization</td>
<td>Add initialization (78%)</td>
</tr>
<tr>
<td>Permission bypass (21.9%)</td>
<td>Sensitive operations without perm check</td>
<td>Add permission check (59%)</td>
</tr>
<tr>
<td>Use-after-free, double-free (4.3%)</td>
<td>Use freed pointer</td>
<td>Add nullification (32%)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(See II. BACKGROUND)
### Modeling different types of security bugs

<table>
<thead>
<tr>
<th>Security operation</th>
<th>Patched version</th>
<th>Unpatched version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer nullification</td>
<td>$\text{FLAG}_{\text{CV}} = 1$</td>
<td>$\text{FLAG}_{\text{CV}} = 0$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{FLAG}_{\text{CV}} = 1$</td>
<td>$\text{FLAG}_{\text{CV}} = 0$</td>
</tr>
<tr>
<td>Permission check</td>
<td>$\text{FLAG}_{\text{CV}} = 1$</td>
<td>$\text{FLAG}_{\text{CV}} = 0$</td>
</tr>
<tr>
<td>Bound check</td>
<td>$\text{CV} &lt; \text{UpBound}$, or $\text{CV} &gt; \text{LowBound}$</td>
<td>$\text{CV} \geq \text{UpBound}$, resp. $\text{CV} \leq \text{LowBound}$</td>
</tr>
</tbody>
</table>

Constraints for security operations from patches. $\text{Flag}_{\text{CV}}$: Flag symbol; CV: critical variable; UpBound: checked upper bound; LowBound: checked lower bound.
## Modeling different types of security bugs

<table>
<thead>
<tr>
<th>Security rules</th>
<th>Patched version</th>
<th>Unpatched version</th>
</tr>
</thead>
<tbody>
<tr>
<td>No use after free</td>
<td>FLAG(_{CV}) = 0</td>
<td>FLAG(_{CV}) = 1</td>
</tr>
<tr>
<td>Use after initialization</td>
<td>FLAG(_{CV}) = 0</td>
<td>FLAG(_{CV}) = 1</td>
</tr>
<tr>
<td>Permission check before sensitive operations</td>
<td>FLAG(_{CV}) = 0</td>
<td>FLAG(_{CV}) = 1</td>
</tr>
<tr>
<td>In-bound access</td>
<td>CV $\geq$ MAX, or/and CV $\leq$ MIN</td>
<td>CV &lt; MAX, resp. CV &gt; MIN</td>
</tr>
</tbody>
</table>

Constraints from security rules. Flag\(_{CV}\) : Flag symbol; CV: critical variable; MAX: maximum bound of the buffer; MIN: minimum bound of the buffer
Generality of patch model

● The existence of three key components in vulnerabilities
  ○ 77% vulnerabilities contains all of three key components
  ○ 11% vulnerabilities contains part of key components

● After extending, SID can support the security-impact determination for them (See VII. DISCUSSION)
What is the common model of patches for security bugs?
Common patch model and key components

// Unpatched program

Vulnerable_operation(Critical variable, ...);
Common patch model and key components

// Unpatched program

Vulnerable_operation(Critical variable, ...);

Violate security rules
Common patch model and key components

// Unpatched program

Vulnerable_operation(Critical variable, ... ) ;

Violate security rules  Security impacts
Common patch model and key components

// Patched program
Security_operation(Critical variable, ...);

Vulnerable_operation(Critical variable, ...);

Violate security rules  \rightarrow  Security impacts
Common patch model and key components

// Patched program
Security_operation(Critical variable, ...);
Common patch model and key components

// Patched program
Security_operation(Critical variable, ...);

Vulnerable_operation(Critical variable, ...);

+ Fix

Violate security rules

Securty impacts

NOT Violate security rules