Detecting Format String Bugs

Consider the following source code:

```c
void loguser(char * user)
{
    char buf[2048];
    snprintf(buf, sizeof(buf), user);
    ..... 
    ..... 
}
```

To understand the format string bug in the above code, consider the stack diagram.

Note:

ARGP: initially points to the third argument in `snprintf`. This is the place where there can be additional optional arguments.

Counter: Keeps track of how many bytes had been printed out so far. This is also a local variable of `snprintf` function.
Attacker provides user = “ XXXX  <0x12345E78>  %0x12345D78d  %n  < shell code>”

Format String Bug Detection and Prevention

Solutions:

1. Educate programmers
   - Mistakes can still occur

2. Automatic Code Auditing
   Source code is fed into the automatic auditor.

   ![Diagram of source code, automatic auditor, warnings, and programmer](image)

   New Code ← Fix

   The problem with this approach is that there can be too many bugs and we will not be able to manually fix all the bugs.

3. Automatic Bug Fixing

   ![Diagram of buggy source, fixer, and bug-free code](image)
From the paper: “Detecting Format String Vulnerabilities With Type Qualifiers”
By: Shankar, Talwar, Foster, Wagner.

Idea: The paper presents a system to automatically detect format string bugs at run time.

1. This system uses static, type theoretic analytic techniques.
2. Data flow analysis

Consider the following code segment:

**Code 1**

```c
int printf(char * fmt, …);
char * getenv(char *);

int main()
{
    char *s, *t;
    s = getenv(“Foo”);
    t = s;
    printf(t);  // safe way to do this : printf(“%s”, t);
}
```

**Code 2:**

```c
char *s = “Hey”;
printf(s);
```

The above Code 2 is considered safe because no dangerous input flows to the format arg of printf.

**Bug Finding Tools**

1. Requires annotations (Consider Code 1)

```
getenv_ret  X   s  X   t  X   fmt  
ref     X   ref  X   ref  X   ref
         |
Stainted char  E   char  E   char  E   Suntainted char
```

So clearly there is a contradiction at the lowest level, since a char can’t be both tainted and untainted.
Contradiction => Potential Error

- Vector < Object
- $untainted < $tainted
- $untainted char < $ tainted char

- False Positive – Warnings but no bugs
- False Negative – Bugs, but no warnings.
- Complete - No false positive
- Precise - Less false positive
- Sound - No false negative

Cqual results

Cqual in theory is sound.
Tools used usually have any two of the three properties:
1. Sound.
2. Complete.
3. Terminate.

Cqual is both sound and terminates.

<table>
<thead>
<tr>
<th></th>
<th>Warnings(Cqual)</th>
<th>Warnings 2(Diff Tool)</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. muh</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2. cfengine</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3. bftpd</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>19</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Current state of the tool : 20 % false rate.

Points to ponder:
- Attacker needs to find 1 bug to win.
- Defender needs to find all the bugs to win.