Venerable Variadic Vulnerabilities Vanquished

Priyam Biswas*  
Stijn Volckaert§

Alessandro Di Federico*†
Yeoul Na§

Scott A. Carr*  
Michael Franz§

Prabhu Rajasekaran§
Mathias Payer*

*Purdue University  
† Politecnico di Milano  
§University of California, Irvine
Variadic Function

- C and C++ support variadic functions
- Variable number of arguments
- Implicit contract between caller and callee
- Cannot statically check the argument types

```c
int add(int n, ...) {
    va_list list;
    va_start(list, n);
    for (int i=0; i < n; i++)
        total=total + va_arg(list, int);
    va_end(list);
    return total;
}

int main(int argc, const char * argv[]) {
    result = add(3, val1, val2, val3);
    result = add(2, val1, val2);
    return 0;
}
```
Motivation

- Parameters of variadic functions cannot be statically checked
- Attacks violate the implicit contract between caller and callee
  - Attacks cause disparity: more/less arguments or wrong argument type
- Existing defenses do not prevent such attacks
## Prevalence of Variadic Functions

<table>
<thead>
<tr>
<th>Program</th>
<th>Call Sites</th>
<th>Functions</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Indirect</td>
<td>Total</td>
</tr>
<tr>
<td>Firefox</td>
<td>30,225</td>
<td>1,664</td>
<td>421</td>
</tr>
<tr>
<td>Chromium</td>
<td>83,792</td>
<td>1,728</td>
<td>794</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>189,908</td>
<td>7,508</td>
<td>1,368</td>
</tr>
<tr>
<td>Apache</td>
<td>7,121</td>
<td>0</td>
<td>94</td>
</tr>
<tr>
<td>CPython</td>
<td>4,183</td>
<td>0</td>
<td>382</td>
</tr>
<tr>
<td>Nginx</td>
<td>1,085</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>4,072</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>Wireshark</td>
<td>37,717</td>
<td>0</td>
<td>469</td>
</tr>
</tbody>
</table>
Threat Model

- Program contains arbitrary memory corruption

- Existing defense mechanisms such as DEP, ASLR, CFI are deployed

- Capabilities of the attacker
  - Directly overwriting the arguments of a variadic function
  - Hijacking indirect calls and call variadic functions over control-flow edges
Control Flow Integrity (CFI)

- Verifies indirect control flow transfers based on statically determined set
- Allows all targets with the same prototype

int foo (int n, ...)
int baz(int n, ...)
int bar(int n, ...)
int boo(n)
void func(int n, ...)
Void func2(int n, ...)
<table>
<thead>
<tr>
<th>Intended target</th>
<th>Actual target</th>
<th>LLVM-CFI₁</th>
<th>pi-CFI₂</th>
<th>CCFI₃</th>
<th>VTV₄</th>
<th>CFG₅</th>
<th>HexVASAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prototype</td>
<td>Addr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Taken</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variadic</td>
<td>Same</td>
<td>Yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>X</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Different</td>
<td>Yes</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Non-Variadic</td>
<td>Same</td>
<td>Yes</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Different</td>
<td>Yes</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Original</td>
<td>Overwritten Arguments</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>√</td>
</tr>
</tbody>
</table>

1. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, USENIX Security 2014
2. Per-Input Control-Flow Integrity, CCS 2015
3. CCFI: Cryptographically Enforced Control Flow Integrity, CCS 2015
4. GCC 6.2 Virtual Table Verification
5. Microsoft Corporation: Control Flow Guard (Windows)
Our Approach

- Enforce contract between caller and callee
- Verify argument types at runtime
- Abort if there is an error
HexVASAN Design

```
void foo ( ...) {
    x = va_arg(int);
    ...
    z = va_arg(char*);
}
```

```
int a, b;
foo(a, b);
```

```c
arg. count = 2
def(arg[0].type = int)
def(arg[1].type = int)
```

```c
data
    MetaData Storage
    read
    check_arg (0, int)
    check_arg (1, char*)
```

```c
caller
    arg
    record
```

```c
callee
    arg
    record
```

```c
OK

?```
Implementation

- Implemented as LLVM pass
- Statically instrument code
- Dynamically verify types of variadic arguments (library)
Real Code Is Hard!

- Handling multiple va_list
  - HexVASAN supports it by recording each va_list separately

- Floating-point arguments
  - Handles floating point and non-floating point arguments separately

- Handling aggregate data types
  - Caller unpacks the fields if arguments fit into registers
  - Traces back to get the correct data type
Evaluation

- Comparison with state-of-the-art CFI mechanisms
- Usage of variadic functions in existing software
- Performance overhead in SPEC CPU2006 benchmark & Firefox
Exploit Detection

- Format string vulnerability in “sudo” CVE-2012-0809
- Attacker can escalate the privileges
- Not detected by -Wformat
- HexVASAN detects exploit

Error: Type Mismatch
Index is 1
Callee Type: 43 (32-bit integer)
Caller Type: 15 (Pointer)
Backtrace:
[0] 0x4019ff <_vasan_backtrace+0x1f> at test
[1] 0x401837 <_vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
Performance Overhead: SPEC CPU2006

Native
HexVASAN
Interesting Cases: Spec CPU2006

- **Omnetpp**
  - **Caller**: NULL
  - **Callee**: char*

- **Perlbench**
  - **Caller**: Subtraction of two char pointers (64 bit)
  - **Callee**: int (32 bit)
## Performance Overhead: Firefox

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>HexVASAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octane</td>
<td>AVERAGE 33,824.40</td>
<td>33717.40</td>
</tr>
<tr>
<td></td>
<td>STDDEV 74.96</td>
<td>125.89</td>
</tr>
<tr>
<td></td>
<td>OVERHEAD 0.32%</td>
<td></td>
</tr>
<tr>
<td>JetStream</td>
<td>AVERAGE 194.86</td>
<td>193.68</td>
</tr>
<tr>
<td></td>
<td>STDDEV 1.30</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>OVERHEAD 0.61%</td>
<td></td>
</tr>
<tr>
<td>Kraken</td>
<td>AVERAGE 885.52</td>
<td>887.12</td>
</tr>
<tr>
<td></td>
<td>STDDEV 11.02</td>
<td>7.31</td>
</tr>
<tr>
<td></td>
<td>OVERHEAD 0.18%</td>
<td></td>
</tr>
</tbody>
</table>
Sample Findings: Firefox

- **Case 1**
  - **Caller**: unsigned long
  - **Callee**: unsigned int

- **Case 2**
  - **Caller**: Bool
  - **Callee**: unsigned long

- **Case 3**
  - **Caller**: void*
  - **Callee**: unsigned long
Conclusion

- HexVASAN successfully monitors variadic arguments
- Detects bugs due to type mismatch in variadic functions
- Negligible overhead in SPEC CPU2006 and Firefox
- Open Source at https://github.com/HexHive/HexVASAN
Thank you! Questions?

Open Source at https://github.com/HexHive/HexVASAN
```c
int add(int n, ...) {
    va_list list;
    va_start(list, n);
    for (int i=0; i < n; i++)
        total = total + va_arg(list, int);
    va_end(list);
    return total;
}

int main(int argc, const char * argv[]) {
    result = add(3, val1, val2, val3);
    return 0;
}
```