Interprocedural Binary Analysis
For Automation and Computer-Human Collaboration

Dr. Matt Revelle
HACR Lab
Montana State University
Bozeman, MT USA

2024 April 01
Outline

• Vulnerability Research Example
• Programs as Graphs
• Simplifying Graphs with Constraints
• Type Inference for Binary Analysis
Interprocedural Binary Analysis Example

```c
cgiFormString(0x1aaa4, &var_910, 0x200); // {"user"
cgiFormString(0x1aaac, &var_710, 0x200); // {"passwd"
cgiFormString(0x1acc, &var_510, 0x200); // {"start"
cgiFormString(0x1acd4, &var_310, 0x200); // {"count"
if (data_2c2e4 != 0)
{
    sub_1194c("name --> [%s][%s]\n", &var_910);
    sub_1194c("count -->[%s][%s]\n", &var_510);
}
int32_t r0_9;
if (sub_160a8(&var_910, &var_710) == 0)
{
    r0_9 = sub_119a4();
} else
{
    sprintf(&var_110, "sqlsearch -t video -o %s -r %s,%m" , "/tmp/video_list.xml", &var_510, &var_310);
    if (data_2c2e4 != 0)
    {
        sub_1194c("cmd[%s]\n", &var_110);
    }
    system(&var_110);
}
```

Authentication check

Command injection

https://www.exploit-db.com/exploits/43435
Interprocedural Binary Analysis Example

```c
if (((uint32_t)*(int8_t*)arg2) != 0)
{
    __b64_pton(arg2, &var_1090, strlen(arg2));
    if (data_2c2e4 != 0)
    {
        sub_1194c("pwd [%s]\n", arg2);
        sub_1194c("pwd decode[%s]\n", &var_1090);
    }
}
int32_t r0_4 = strcmp(arg1, "mydlkinBRionyg");  
int32_t r0_6;  
void* r0_7;  
if (r0_4 == 0)
{
    r0_6 = strcmp(&var_1090, "abc12345cba");
    if (r0_6 == 0)
    {
        r0_7 = 1;
    }
}
```

Decode Base64 encoded password

Hard-coded login credentials

`is_authenticated` variable

https://www.exploit-db.com/exploits/43435
Interprocedural Binary Analysis

Example

- Task: Find a feasible path that uses the hard-coded credentials and reaches the vulnerability
- Manual tracking of feasible paths and constraints over multiple function control-flow graph (CFGs)
Interprocedural Binary Analysis

Example

• Number of paths in a function can be very large, but often many are infeasible

• Automated removal of these paths can have a big impact

• Can use automated analyses to automatically simplify an interprocedural CFG as it is constructed
Interprocedural Binary Analysis

Problem Statement

Use automated analyses to interactively help vulnerability researchers manage the complexity of analyzing program binaries for vulnerabilities.
Blaze
Static Analysis Framework

• Built around *interprocedural control-flow graphs (ICFGs)* and a typed intermediate language *(PIL)*

• Supports symbolic analysis through satisfiability modulo theories (SMT) solvers

• Open source, written in **Haskell**

• Support for many executable formats and architectures via **Binary Ninja** and **Ghidra**
What is a Program Executable?

- ELF header
- Program header table
- .text
- .rodata
- ... (omitted for brevity)
- .data
- Section header table

![Binary code representation](image-url)
Programs as Graphs

def funcA(x, y):
    result = None
    if x == y:
        result = 2 * x
    else:
        result = x + y
    return result

def funcB(xs):
    result = 0
    for x in xs:
        result += x
    return result
Control-Flow Graphs (CFGs)

```c
long bar(long a) {
    if (a == 0) {
        printf("Error: a can't be zero.");
        return 0;
    }
    else {
        return a * a;
    }
}
```
Interprocedural Control-Flow Graphs (ICFGs)

- Control-flow graphs (CFGs) that may span across function calls
- In ICFGs, function calls are expandable **call nodes**
- ICFGs can be constructed programmatically or by user interaction
Dominators
Influence of a Node

- A node $x$ in a control-flow graph **dominates** node $y$ if every path from the root to $y$ passes through $x$
- A node may have many dominators
Branch Contexts

Dominating Constraints

• Nodes dominated by a conditional branch are in a **branch context**

• Every branch context is associated with a constraint

• Branch contexts can be nested

• Use branch contexts to determine if a node is reachable

\[
\begin{align*}
\text{arg1} \neq 0 & \land \var_10 \neq 0 & \land \var_10 = \text{arg1} \\
\text{arg1} = 0 & \\
\var_10 = 0 & \land \var_10 = \text{arg1}
\end{align*}
\]
Constraint-Driven Transformations

Call Expansion

Unsatisfiable
Constraint-Driven Transformations

Call Expansion

- The call to bar is expanded
- Infeasible path is automatically removed from the ICFG
Constraint-Driven Transformations
CVS Example
Satisfiability Modulo Theories (SMT)

- **SMT solvers** can check if a formula is satisfiable
- Support for integers, floats, bit vectors, arrays, and more through theories
- Describe program constraints as a mathematical formula
- Behind the scenes in Blaze, typed PIL statements are used to generate SMT formulas

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (assert (< x 10))
4 (assert (> y 0))
5 (assert (= x (* y 2)))
6 (check-sat)
7 (get-model)
```

```
sat
 (define-fun y () Int 1)
 (define-fun x () Int 2)
)
TABLE I. THE TYPES AND TYPE CONSTRUCTORS AVAILABLE IN THE PIL TYPE SYSTEM. SINGLE-LETTER METAVARIABLES ARE USED AS PLACEHOLDERS FOR UNSPECIFIED VALUES (LOWERCASE) AND TYPES (UPPERCASE).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitVec w</td>
<td>Bit vector types specified by bit-width w.</td>
</tr>
<tr>
<td>Int w s</td>
<td>Integer types specified by bit-width w and signedness flag s.</td>
</tr>
<tr>
<td>Pointer w T</td>
<td>Pointer types specified by the pointer bit-width w and pointee type T.</td>
</tr>
<tr>
<td>Float w</td>
<td>Float types specified by bit-width w.</td>
</tr>
<tr>
<td>Char w</td>
<td>Character types specified by bit-width w.</td>
</tr>
<tr>
<td>Bool</td>
<td>The Boolean type.</td>
</tr>
<tr>
<td>Record m</td>
<td>Record types specified by a map m of field offsets to field types.</td>
</tr>
<tr>
<td>Array n T</td>
<td>Array types specified by a length n and element type T.</td>
</tr>
<tr>
<td>CString n</td>
<td>Null-terminated sequence of Char 8 values specified by length n.</td>
</tr>
<tr>
<td>Function R [P]</td>
<td>Function types specified by return type R and variable number of parameter types [P].</td>
</tr>
<tr>
<td>()</td>
<td>The Unit type inhabited by the single value ().</td>
</tr>
<tr>
<td>T, ⊥</td>
<td>The Top and Bottom types.</td>
</tr>
</tbody>
</table>
Type Inference

A

\&\text{var}_20 : T_1
T_1 <= \text{Pointer 32} T_2
var_20 : T_3
T_2 = T_3
T_3 <= \text{BitVec 32}

B

r0\#1 : T_4
T_1 = T_4

C

\text{strtol} : T_5
T_5 <= \text{Function} T_6 [T_7 T_8 T_9]
T_4 = T_7
T_7 <= \text{Pointer 32} T_{10}
T_{10} <= \text{CString ?}
T₂ and T₃ are the same type.
T₁, T₄, and T₇ are all the same type. Simplify the rules to be expressed in terms of T₂ and T₁.

T₂ and T₁₀ must have the same type.

So then T₂ must be a subtype of CString.

We saw before that T₂ is a 32-bit bit vector. Unify that type with CString.
Blaze
Static Analysis Framework

• Built around **interprocedural control-flow graphs (ICFGs)** and a typed intermediate language (**PIL**)

• Supports symbolic analysis through satisfiability modulo theories (SMT) solvers

• Open source, written in **Haskell**

• Support for many executable formats and architectures via **Binary Ninja** and **Ghidra**

"Haskell logo." https://www.haskell.org/img/haskell-logo.svg
"Binary Ninja logo." https://www.cyberus-technology.de/assets/images/products/tycho/loop_binary_ninja.png
"Ghidra logo." https://ghidra-sre.org/images/GHIDRA_1.png