Homework 3 (Alternative)
Posted Monday February 14, Due Monday February 28
50 points

1. Overview

In this homework you will build a classical context-insensitive, flow-insensitive points-to analysis over the PCode IR provided by the Ghidra reverse engineering tool. Ghidra reverses x86 (and other architectures) into a higher-level three-address code IR called PCode. Given an x86 binary as input, you will run Ghidra to reverse (i.e., decompile) the binary, then run your points-to analysis on the PCode IR. We will build a twist on the classical points-to analysis: certain memory locations will be marked as tainted and points-to analysis will track the flow of tainted locations through the program.

The result will be a points-to graph that represents memory relations. You will present a visualization of the graph using the Ghidra API. Specifically, we will be looking at two kinds of relations:

1. A pointer \( p \) used at an indirect call statement \( \text{CALLIND } p \), stores user input. This pattern indicates a potential vulnerability: user input may flow to the pointer used at the indirect call; thus, a malicious user may supply a function that hijacks program execution.

2. A location \( \text{loc} \) used as an argument to an output statement (typically, but not exclusively \( \text{printf(loc)} \)), stores the address of a library function. This represents a memory leak that (combined with other vulnerabilities) may lead to a malicious user hijacking program execution.

We saw these patterns of vulnerabilities over and over in MBE. In this project, you will explore whether classical techniques such as points-to and taint analysis built on top of the PCode IR, can reason about such vulnerabilities effectively.

2. Getting Started with Ghidra

Download and install Ghidra: https://ghidra-sre.org/

2.1. Reversing a Binary. Since you are new to PCode, start with a binary compiled with debugging information (this will create a higher-quality PCode to work with in the beginning). Start with the MBE binaries, which we know have vulnerabilities, specifically the Heap chapter ones. For example, I compiled \(08\_\text{level}\_3\_c\) (on an x86 Mac) using the following command:

\[
gcc -o 08\_\text{level}\_3 \ -g \ -fpie \ -pie \ 08\_\text{level}\_3\_c
\]

Start a new project and Import the binary, \(08\_\text{level}\_3\) in this case. In the Program Tree window (upper left corner) of the Code Browser tool you will see a list of functions. If you double click on a specific function, e.g., \(\text{main, sshAction}\), you will see the C-like decompiled code for that function in the Decompile window in the upper right corner. Take some time to familiarize yourself with the environment, views and functionality of the Code Browser tool.

Click on the green arrow in the tool bar to get to the Scripting environment. There are many sample scripts. The most useful ones for our purposes are the PCode scripts. I have written a starter PCode script https://www.cs.rpi.edu/~milanova/csci4450/PtAnalysis.java that
traverses the functions in the binary and prints each PCode instruction. Your task is to translate PCode instructions into appropriate constraints then solve these constraints to build a points-to graph (in the same fashion of constructing and solving constraints as in HW2).

2.2. **Useful Classes.** The Ghidra API is at https://ghidra.re/ghidra_docs/api/. Class `Varnode` represents a named memory location. Varnodes are used as operands in the 3-address PCode instructions. Class `PCodeOp` represents PCode instructions, and so on. There is representation for control flow graphs, basic blocks, edges, etc. `HighFunction` represents a high-level PCode-reversed function and interface `Function` looks into the low-level (Assembly) Ghidra representation of a function. There are many other useful classes and interfaces.

3. **Points-to Analysis over PCode**

As mentioned already, the gist of this homework is to (1) create the appropriate constraint classes, (2) translate PCode instructions into constraints, (3) code a worklist-like algorithm to solve the constraints, and (4) visualize the solution. The workflow is very similar to the one in HW2, however, translation of PCode instructions is a lot more challenging! Below I have written some nodes on the translation of PCodes into analysis statements and constraints.

3.1. **Four Kinds of Statements and their Interpretation.** The table below defines the abstract program statements and their interpretation. The most challenging task is to map each PCode instruction into one or more of these kinds of statements. Then you can create constraints for each statement as specified.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Address taken $p = loc$</td>
<td>${loc} \subseteq pts(p)$</td>
</tr>
<tr>
<td>(2) Assign $p = q$</td>
<td>$pts(q) \subseteq pts(p)$. If $q$ is tainted, then $p$ is tainted.</td>
</tr>
<tr>
<td>(3) Load $p = *q$</td>
<td>For each $y \in pts(q)$, $pts(y) \subseteq pts(p)$; if $y$ is tainted, then $p$ is tainted.</td>
</tr>
<tr>
<td>(4) Store $*p = q$</td>
<td>For each $x \in pts(p)$, $pts(q) \subseteq pts(x)$; if $q$ is tainted, then $x$ is tainted.</td>
</tr>
</tbody>
</table>

A location can be tainted in green or in red. It is tainted green if it may store important information such as a libc function address. It is tainted red if it may store user input.

3.1.1. **Address taken.** Lifting PCode instructions into Address taken statements is a challenge. For example, consider the following PCode sequence:

```
(unique, 0x3100, 8) INT_ADD (register, 0x20, 8) , (const, 0xffffffffffffffa8, 8)
--- CALL (ram, 0x100001540, 8) , (unique, 0x3100, 8) , (const, 0x40, 4)
varnode1 LOAD varnode2 , (unique, 0x3100, 8)
```

It roughly corresponds to the following C-like sequence:

```c
char local_58[4];
readData(local_58,0x40);
char c = local_58[0];
```

The program allocates space for the buffer then passes the buffer to the `readData` function. `readData` subsequently calls library function `read`, which stores user input into the `local_58` buffer. Looking back at the PCode, the load statement out of `(unique, 0x3100, 8)` reads a tainted character.
The analysis will need to account for the implicit allocation of the buffer on the stack; failing to do so may miss points-to edges and taints. In our example the analysis can record an address-taken constraint, say

\[(\text{const, 0xffffffffffffffa8, 8}) \subseteq pts((\text{unique, 0x3100, 8}))\]

Eventually location (\text{const, 0xffffffffffffffa8, 8}) becomes tainted due to the user input. The constraint for varnode1 LOAD varnode2 , (\text{unique, 0x3100, 8}) retrieves location (\text{const, 0xffffffffffffffa8, 8}), i.e., where (\text{unique, 0x3100, 8}) points to; there are no new points-to edges, however, the red taint propagates to varnode1.

In addition to arrays on the stack (known as buffers), you will need to account for function addresses, local addresses and heap allocations. For example, each malloc call will give rise to a location in the analysis and a corresponding constraint that links the location and the left-hand-side of the malloc statement.

3.1.2. Loads and Stores. There are explicit LOAD and STORE PCodes and their constraints will be as expected.

3.1.3. Assign. For the most part, you can adopt a conservative approach and reduce PCodes to assignments statements. Namely, you can consider an assignment \(\text{Output} = \text{Input}[i]\) for each \(\text{Input}[i]\) in the PCode, each giving rise to an assignment constraint. Note that CALL PCodes will require handling to account for implicit assignments of actual arguments to formal parameters and return values to left-hand-side variables. This is likely non-trivial as you will need to figure out what PCode makes of the x86 calling conventions.

3.2. Summaries for Library Functions. Another challenge is the handling of library functions. You do need to account for a subset of library functions: \text{strcpy}, \text{memcpy}, \text{memset}, \text{read}, \text{scanf}, \text{strtoul}, etc.. These are functions that may introduce/propagate points-to edges or taints. Another set of functions you need to account for is “sink” functions, e.g., \text{printf}.

3.3. Benchmarks. You will run the analysis on the three challenges from the Heap Wargames chapter: 08_level_1, 08_level_2, and 08_level_3. It will be interesting to see how effective (or how ineffective!) the analysis is in identifying vulnerabilities. If the analysis works well, we can run on other benchmarks such as the DARPA Cyber Grand Challenge ones!