Bouncer: Securing Software by Blocking Bad Input

Sathish Kuppuswamy
&
Yufei Fu
Department of computer Science
University of Texas at Dallas
March 21st, 2012
Outline

- Bouncer
- Existing Techniques
- Bouncer Techniques
- Final Filter
Bouncer

- A system that prevents attacks by dropping exploit messages before they are processed by a vulnerable program.

- Type safe languages like C++, java - to ensure type safety (throws exception)
  - CRED (C Range Error Detector) - to detect buffer overflows
    - leads to loss of data and denial of service (restarting program)

- Bouncer improves the reliability and availability of programs under attack
Techniques

Bouncer introduces three practical techniques to generalize the initial filter to block additional exploits of the same vulnerability:

- **Precondition slicing** is a new form of program slicing that uses a combination of static and dynamic analysis to remove unnecessary filter conditions.

- **Symbolic summaries** generalize the conditions captured by the symbolic execution inside common library functions.

- **Generation of alternative exploits** guided by symbolic execution.
Architecture

1. Sample exploit
2. Vulnerable program instrumented to detect attacks
3. Trace
4. Precondition slicing and symbolic summaries
5. Filter conditions
6. Combine filter conditions
7. Generation of alternative exploits
8. Filter conditions
9. Yes/no exploit
10. New exploit?
DFI (Data Flow Integrity) to detect attacks

- For each value read by an instruction in the program text, DFI uses static analysis to compute the set of instructions that may write the value.

- At runtime, it maintains a table with the identifier of the last instruction to write to each memory location.

- The program is instrumented to update this table before writes, and reads are instrumented to check if the identifier of the instruction that wrote the value being read is an element of the set computed by the static analysis.

- If it is not, DFI raises an exception.
• If the sample exploit is valid, the execution trace is sent to the module that runs the precondition slicing algorithm.

• **Vulnerability point** - We call the instruction where the attack is detected.

• **Nirvana** to generate an execution trace.

• **Vigilante technique** to generate filters automatically.
  - This technique replaces the concrete value of each byte in the sample exploit by a symbolic value $b_i$ and performs forward symbolic execution along the trace of x86 instructions. It adds a condition to the filter for each branch that depends on the input.
Example

ProcessMessage(char* msg) {
    char buffer[1024];
    char p0 = 'A';
    char p1 = 0;
    if (msg[0] > 0)
        p0 = msg[0];
    if (msg[1] > 0)
        p1 = msg[1];
    if (msg[2] == 0x1) {
        sprintf(buffer, "servers\%s\%c", msg+3, p0);
        StartServer(buffer, p1);
    }
}

Sample exploit message:
Message starts with three bytes equal to 0x1 followed by 1500 non-zero bytes and byte zero. Processing this message causes DFI to throw an exception when p1 is accessed to set up the call stack for StartServer because p1 has been overwritten.

\[ b_0 > 0 \land b_1 > 0 \land b_2 = 1 \land b_{1503} = 0 \land _{2 \leq i \leq 1503} b_i \neq 0 \]
Precondition slicing:

- **Static Vs Dynamic:**
  - **Static analysis** - very imprecise when applied to programs with pointers. They tend to classify most instructions as relevant.
  - **Dynamic analysis** - use dynamic information to improve precision. They take an input, generate an execution trace, and use the dynamic dependencies observed during the trace to classify instructions as relevant.
    - may introduce **false positives**.
    Dynamic slicing can lead to the removal of necessary conditions from the filter because it does not capture dependencies on instructions that were not executed in the trace.
Example - false positive

```c
int a = 0, b = 0;
int *c = &b;
if (msg[0] == 'a') a = 1;
if (msg[1] == 'a') c = &a;
*c = 0;
if (a)
Vulnerability();
sample exploit msg = "ab"
Valid input msg="aa"
```
Pseudo-code for the slicing algorithm

ComputeSlice() {
    while (!trace.IsEmpty) {
        cur = trace.RemoveTail();
        if (cur.IsRet) {
            call = trace.FindCall(cur)
            if (MayWriteF(CalledFunc(call), live))
                Take(cur);
        } else if (cur.IsCall) {
            Take(cur);
            foreach (e in trace.CallArgSetup(cur)) {
                Take(e);
                trace.Remove(e);
            }
        } else if (cur.IsBranch) {
            if (!Postdominates(slice.head, cur)
                || WrittenBetween(cur, slice.head)) Take(cur);
        } else {
            if (MayWrite(cur, live)) Take(cur);
        }
    }
}

void Take(cur) {
    slice.AddHead(cur);
    live.UpdateWritten(cur);
    live.AddRead(cur);
}
Data Structures:

- **cur** is the trace entry being processed.

- **slice** is a list of trace entries that were added to the path slice. Initially, it contains the entry for the vulnerability point instruction.

- **live** keeps track of dependencies for instructions in slice. It contains entries for operands read by these instructions that have not been completely overwritten by instructions that appear earlier in the trace.
  - Entries in live contain a pointer to the corresponding operand in the code, the register or memory address from which the instruction read the operand in the execution trace, and the symbolic or concrete value of the operand read by the instruction in the symbolic execution.
  - Entries also keep track of portions of the operand that have been overwritten by instructions that appear earlier in the trace. Initially, **live contains the operands read by the instruction at the vulnerability point**.
MustAlias:

Assume that p0 and p1 point to the same storage location and that this fact cannot be determined by the static analysis.

Precondition slicing can remove the condition $b_0 > 0$ from the initial filter.

When $p1 = msg[1]$ is processed, the operand for *p0 is removed from live because its storage location is overwritten. Therefore, the branch that checks $msg[0] > 0$ is not added to the slice.
MayAlias:

Assume that $p_0$ and $p_1$ point to different locations but static analysis cannot determine this fact.

Precondition slicing can remove the condition $b_1 > 0$.

* $p_1 = \text{msg}[1]$ is not taken because it does not overwrite any operand in live and $p_1$ is in live. So the branch that checks $\text{msg}[1] > 0$ is not taken.

$$b_0 > 0 \land b_1 > 0 \land b_2 = 1 \land \forall_{2 < i < 1016} b_i \neq 0$$

$$b_2 = 1 \land \forall_{2 < i < 1016} b_i \neq 0$$
Symbolic summaries

- Precondition slicing is not effective at removing conditions added by instructions inside library functions.

- Knowledge about the semantics of common library functions to generate symbolic summaries that characterize the behavior of a function as a set of conditions on its inputs.

2 cases
  - Vulnerability point is inside a library function
  - Library function is called in the path towards the vulnerability.
In the first case, the symbolic summary is simply a condition on the arguments of the function that is true exactly when the vulnerability can be exploited.

\[ b_0 > 0 \land b_1 > 0 \land b_2 = 1 \land \text{memcpy}(dst, src, n) \land b_i \neq 0 \quad \text{for} \quad 2 < i < 1016 \]

In the second case of symbolic summary for library functions that are called in the path towards the vulnerability.

\[
\text{if (stricmp(s, "A string") == 0) Vulnerability();}
\]

the vulnerability in this example is reachable if the attacker supplied string equals “A string” after both are converted to lowercase. The conditions that we extract automatically from a sample execution of `stricmp` will only capture a particular value of `s` that satisfies the comparison.

\[
(s[0] = A \lor s[0] = a) \land \ldots \land (s[8] = G \lor s[8] = g) \land s[9] = 0
\]
Search Strategy for other attacks:

- **DART** *(Directed automated random testing).*
  
  It takes a prefix of the conditions negates the last one and feeds the resulting conditions to a constraint solver to obtain new test inputs.

  Find all exploits of the same vulnerability but time consuming.

- Another approach generates alternative exploits by removing or duplicating bytes in the original exploit messages.

  Not guaranteed to find all exploits of the same vulnerability but it is simple and fast.

  Pick the bytes with the lowest scores because they are likely to be filler bytes in buffer overflow exploits.

**Scoring:**

Give a score to each condition equal to the total number of bytes in conditions divided by the number of bytes that have an identical condition. Each byte has a score equal to the sum of the scores of the conditions it appears in.
Final Filter

- Taking the disjunction of all filters can result in a final filter with high overhead.

- A common structure is a set of byte indices in the beginning of a message that have the same condition in all filters.

- Typically followed by sequences of byte indices that have different lengths in different filters but have the same conditions applied to each byte in the sequence in each filter.

- They are followed by terminator bytes with the same conditions in each filter.

- The final filter is an x86 executable.