Bouncer: Securing Software by Blocking Bad Input

Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, Marcus Peinado
Problem

- Attackers can exploit software vulnerabilities to crash or seize control of programs.
- Most famously, buffer overflow problems in Windows.
- Obviously a major problem for everyone.
Current Strategies

- Type-safe languages (Java, C#, etc.) have mechanisms to detect and intercept exploits.
- Tools like CRED perform runtime checks on unsafe languages like C.
- Problem:
  - By the time an exploit is detected, program may have to be terminated, leading to potential for DOS attacks and data-loss.
Input Filtering

- Rather than wait for out-of-bounds reference, Bouncer (and predecessors such as Vigilante) identify messages that can lead to exploit and drop them.
- No longer have to restart program.
- Most programs can tolerate message loss.
Basic Technique

- When new exploit is detected by tool such as DFI, it is fed to Bouncer.
- Bouncer creates a filter which will block input from the exploit and other attacks which use the same vulnerability.
New Ideas

- Precondition Slicing removes unnecessary conditions from filter
- Symbolic Summaries for common library functions
- Generation of alternative exploits based on a sample exploit.
Filter Generation

- Assigns a symbolic value to each byte in input message and performs symbolic execution along trace from receipt of input to point of exploit.
- From this, generates set of conditions in input that lead to exploit.
- Removes extraneous conditions in order to generalize filter.
Sample of Vulnerable Code

```c
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);
    } }
```
Generating Conditions for Sample Code

- Initial condition from code:
  \[ b_0 > 0 \land b_1 > 0 \land b_2 = 1 \land b_{1503} = 0 \land \forall_{2<i<1503} b_i \neq 0 \]

- After generalizing for size of buffer:
  \[ b_0 > 0 \land b_1 > 0 \land b_2 = 1 \land \forall_{2<i<1016} b_i \neq 0 \]

- After recognizing first two conditions as unnecessary to exploit:
  \[ b_2 = 1 \land \forall_{2<i<1016} b_i \neq 0 \]
Symbolic Execution

- Defines a total order on instructions of the trace in assembly code
- Identifies storage locations as “Symbolic” if they are dependent on input.
- For every branch that is dependent on a “symbolic” value, adds a condition to the filter.
Precondition Slicing

- Removes conditions from the filter which are not necessary to the exploit.
- This allows the filter to block more attacks which use variants on the same exploit.
Static Slicing

- Many current algorithms for program slicing use static analysis.
- This is problematic in languages with pointers like C and C++, in which target of pointer may not be known at time of analysis.
Dynamic Slicing

- Dynamic slicing algorithms use dependencies observed in execution trace to decide which conditions are relevant.
- May be inaccurate because of dependencies from outside the trace.
- May remove necessary conditions.
- May lead to false positives.
Precondition Slicing

- Combines elements of static and dynamic slicing.
- Generates a “path slice” which is a subsequence of instructions necessary to exploit the vulnerability.
- Uses alias analysis to track pointer values.
- Iterates backward through trace, deciding which instructions to include in slice.
Symbolic Summaries

- To avoid adding all instructions from library functions to the slice, generates symbolic summaries of common library functions.
Search for New Attacks

- Try to automatically generate new attacks based on exploit found in sample attack.
- Iteratively generate filters for new attacks, then combine filter to obtain more generalized protection.
Evaluation

- Evaluated Bouncer using known vulnerabilities in SQL Server, ghhttpd, nullhttpd, and stunnel
- Generated filters for vulnerabilities, then evaluated with attacks.
- No false positives, and blocked all attacks for SQL Server and stunnel
Evaluation Continued

- Filter generation time. A rudimentary filter could be generated and deployed in seconds, but a final filter could take hours to generate (16.2 for stunnel).
- Running time overhead of applying filter is generally linear relative to the length of the message