Name: Vedant Deepak Borkute
SID: 862552981
NetID: vbork001
HW 3

An empirical study of the reliability of UNIX utilities" (fuzzing) :

1. What experience motivate the authors to conduct this more systematic experiment? Please explain what was the source of random inputs in that experience.

The experiment was motivated by an experience where one of the authors was logged onto a workstation via a dial-up line from home during a dark and stormy night. Rain had affected the phone lines, resulting in frequent spurious characters (line noise) on the line. They were surprised that these spurious characters caused programs, including basic operating system utilities, to crash ("core dump"). The source of random inputs in that experience was the frequent spurious characters (line noise) on the dial-up phone lines.

2. What is ptyjig for? How it's different from fuzz? How does it work?

Ptyjig is a program written to test interactive utility programs, such as screen editors like vi or the mail program. Ptyjig is different from fuzz because interactive utilities expect their standard input file to have the characteristics of a terminal device. The standard output from fuzz (a stream of random characters) is not sufficient to test these programs.

Ptyjig works by first allocating a pseudo-terminal file, which i a two-part device file. One side looks like a standard terminal device file (e.g., /dev/ttyp?), and the other side can be used to send or receive characters on the terminal file (e.g., /dev/ptyp?). After creating the pseudo-terminal file, ptyjig starts the specified utility program and passes characters sent to its input through the pseudo-terminal to be read by the utility.

3. What is the most common cause for crash?

The most common cause for crash found during testing was the class of pointer and array errors i.e programs sequentially accessing array cells with a pointer or subscript while failing to check for exceeding the array's range.

EXE: Automatically Generating Inputs of Death:

1. How does EXE symbolically execute a C expression? Check Figure 3 and 4.

When an operation in the checked code uses a symbolic value (i.e., input-derived expressions), EXE does not run the operation. If the symbolic operation is a C assignment (v=e), EXE builds the symbolic expression e_sym representing e and records that the concrete address of v maps to e_sym. Instead of running the operation, the operation is passed to the EXE runtime system, which adds it as an input-constraint for the current path.

2. How does EXE explore different execution paths? Check Figure 5 and 6.

EXE explores different execution paths by forking execution (using the fork system call) when it encounters a conditional check on a symbolic expression. On the true branch, it constrains the expression to be true, and on the false branch, it constrains the expression to be false. Execution paths are also forked at implicit symbolic branch points inserted by the compiler, such as dangerous operations (e.g., checking for a null or out-of-bounds memory reference). If EXE adds a constraint that makes the path impossible, it queries the constraint solver STP to determine that no solution exists, and the execution of that path stops.

3. How does EXE generate "inputs of death?"

EXE generates "inputs of death" by using its constraint solver STP to solve the current path constraints for concrete values when a path terminates or hits a bug. For dangerous operations where EXE detects that the constraints allow a bug (e.g., an out-of-bounds access), EXE forks execution, asserts that the condition occurs on the true path, emits the test case (the input of death), and terminates that path.

A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities" (static analysis) :

1. Compare to that simple static analysis (i.e.,, grep strcat) mentioned in the stack buffer overflow paper, how does the approach proposed in this paper improves the precision and reduces false positives?

Tools like grep produce many false alarms when searching for unsafe string operations. This paper improves precision by formulating buffer overrun detection as an integer range analysis problem, modeling each string buffer with integer ranges for its allocated size and current length. Safety is verified by checking that the allocated size ≥ maximum length. This approach greatly reduces false positives—by an order of magnitude in tools like sendmail 8.9.3—compared to grep.

2. Static analysis trades precision for scalability, give an example of the imprecise modelings (i.e., heuristics) discussed in this paper.

An example of imprecise modelings is Flow-Insensitive Analysis; This analysis ignores control flow and statement order to improve scalability and efficiency. This means that complex operations like strcat() are difficult to model accurately, often leading to every non-trivial use being flagged as a potential buffer overrun.

3. What is the main limitation of the presented approach? How this limitation can be addressed?

The main limitation of the presented approach is the large number of false alarms it produces due to the imprecision inherent in the integer range analysis. This requires significant time and effort from a human auditor to review the warnings. It can be addressed by moving to a more

precise, but slower, analysis, such as a flow-sensitive or context-sensitive analysis. Alternatively, standard analysis techniques like SSA form, Shostak's loop residues, or a points-to analysis could be incorporated to improve the prototype's precision substantially

Bugs as Deviant Behavior:

1. Explain how the null pointer consistency check works.

The null pointer consistency check works by employing internal consistency to find errors by tracking programmer beliefs about a pointer p and flagging contradictions. A belief set (B_p) is associated with each pointer p.

1. A dereference of p implies the MUST belief that p is non-null. An error is flagged immediately if p's belief set currently contains the belief "null".

2. A comparison (e.g., p==NULL) implies that before the check, p's value was unknown (could be null or not null), and flags an error if p was known more precisely (redundancy check).

3. After the comparison, the belief propagates forward: p is constrained to be "null" on the true path and "not null" on the false path.

When paths join, the beliefs are combined using a union operation.

2. Use the statistical lock inference example, explain how to check MAY beliefs.

For the statistical lock inference example, checking MAY beliefs (such as whether a shared variable v is protected by a lock l, which might be a coincidence) involves treating all possible variable-lock pairs (v,l) as MUST beliefs initially. The checker records:

1. Checks (N): The total number of times the variable v was accessed.

2. Errors (C): The number of times v was accessed without lock l held (a violation of the assumed rule).

Then, a statistical analysis is used, typically applying the z statistic, to rank the resulting errors. This ranking measures the ratio of successful checks (E=N−C) to errors (C). Errors derived from beliefs that were observed in the majority of cases (e.g., lock l almost always protecting v) are ranked higher and are considered more credible errors

3. Read the related work section, what is the most basic/common way for developers to express what they want/expect, and for compilers to check if the expectations are violated? You all should have used it before.

The most basic/common way for developers to express what they want/expect, and for compilers to check if expectations are violated, is through language type systems. Type systems

find more bugs on a daily basis than any other approach, requiring programmers to apply a fixed type system throughout their code.