# Automatic generation of patch-based unsafe inputs

September 28, 2012

## 1 Introduction

Managing the security of complex systems poses a great challenge. Incremental software patches are by far the most common means of plugging security holes: be it at the level of the operating system kernel, the middleware or the applications.

Contrary to the view that security patches enhance the security of the software system, a recent work [1]—henceforth refered to as APEG —shows that the former can actually help attackers in exploiting the very same vulnerabilities that were patched by exploiting the vulnerability on the unpatched system. Note that this is nontrivial: a security patch need not reveal the vulnerability in a straightforward way. One would need to analyse the target program to list program variables that are effected by the patch and where they are used. This entails significant work.

Formally, APEG proposes techniques for the automatic patch-based exploit generation problem: given a program $P$, and a patched version of the same program, $P'$, automatically generate an exploit for the potentially unknown vulnerability in $P$ but patched in $P'$. Furthermore, the work targets input validation vulnerabilities only: vulnerabilities arising out of invalid inputs to the program. Hence, the attack can be modelled as a constraint satisfaction problem, with the constraint that an input fail the input sanitisation checks added in $P'$ with an assumption that there is an exploit corresponding to a given input validation vulnerability. APEG argues that given that the delivery of patches is staggered over long time periods, there is a realistic time window in which unpatched systems are vulnerable to attack.

## 2 Proposed Work

In this project, we seek to emulate the work done in [1] in a different setting. More precisely, our setting is different from theirs in the following ways:

1. APEG works on closed-source binaries (executables), while we only consider open-source software. This lets us focus on the problem of auto-

matic exploit generation and not worry about reverse engineering a patch. Specifically, our target of attack would be userspace libraries (written in C language) in the Android software stack e.g., *bionic* libc, dalvik runtime libraries etc.

2. APEG proposes techniques based on dynamic program analysis only, static program analysis only and a mix of both, while we focus on static analysis only.

### 2.1 Methodology

The project could be logically divided into the following phases:

1. Identify what constitutes a security patch for the attack target. This would need to be manually by looking up the commit history (log) of the target software.[1]

2. From the security patches thus identified, filter out those that do NOT address an input validation vulnerability e.g., focus on sanitisation checks added on program variables. The patches that remain are those that will constitute our exploitable set.

3. Using techniques from static analysis, identify which part of the unpatched software has an input validation vulnerability. This phase is going to be challenging becuase it is nontrivial to search for a variable sanitised in $P'$ that is used as an input in $P$. We would be relying upon the CIL infrastructure [2] for obtaining an intermediate representation (Abstract Syntax Tree) of the target software that is amenable to static analysis.

4. Express the problem of finding an exploitable input as a constraint satisfaction problem and rely upon Satisfiability Modulo Theories (SMT) solvers to obtain a solution to the problem.

### 2.2 Deliverables

Building a framework for statically analysing the target software for possible input validation vulnerabilities would be a deliverable. However, the most important deliverable of this project would be (hopefully) a few hitherto unknown input validation vulnerabilities on a small portion (native libraries) of Android's software stack.

## References

[1] D. Brumley, P. Poosankam, D. Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Secu-*

[1] Android maintains a git repository for each component in its software stack. Commits are always tagged with textual comments about changes done.

*rity and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143 –157, may 2008.

[2] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.