TABLE OF CONTENTS

1. IDENTIFICATION & SIGNIFICANCE OF THE OPPORTUNITY

As the use of software to control more of the world inexorably increases, so does the importance of having confidence that software cannot easily be subverted by attackers. To provide this assurance, several techniques have been developed. One of the most effective and low-cost is software fuzzing, which randomly and semi-randomly permutes software inputs (e.g. files or network data). The software being tested is monitored for indicators of poor code quality and security vulnerabilities, such as crashes.

Fuzzing is a very effective technique; however, it is frequently too effective. Fuzzing generally produces hundreds or thousands of crashes, each a candidate software vulnerability that must be mitigated. Many of the crashes will have the same root cause but initially appear to be unique issues. When addressing the results of a fuzzing campaign, three concerns are certain to emerge:

- Which of the crashes identified have the same fundamental cause?
- For a given fundamental cause, what is the minimum input that triggers the crashing condition?
- Which of the crashes identified is the most serious?

These concerns are important because categorization of similar crashes can motivate both the fuzzing campaign itself as well as the remediation efforts following. The severity of each crash is a valuable metric to helps prioritize limited remediation resources.

What if a system could characterize a crash, automatically, in terms of severity?

Currently, methods used in practice to quantify the severity of a crash are ad-hoc. An expert examines the crash to answer the question: "how easily could this be turned into a code execution exploit?" The greater the ease, the greater the urgency associated with developing, testing and deploying a patch. Ideally, the fuzzing framework would have a post-processing component that utilizes all the data available to automatically group crashes with similar fundamental causes and categorize their severity.

1.1 Background

Compiler Theory, Program Analysis, and Formal Methods are very active research fields and have several solutions that we believe will be directly applicable to the problem at hand. In particular, we believe taint tracking, information flow theory, constraint solving, and symbolic execution can be directly applied.

1.1.1 Taint Tracking

Taint tracking is an analysis technique meant to identify specific areas of code that operate on "tainted" data as it flows through an application. A taint tracking system marks some input data as tainted and then updates and propagates that taint during execution. In a security context, taint tracking is an incredibly valuable tool since, at the point of the software vulnerability, you can directly quantify an attacker's level of control.

Taint tracking has been studied simultaneously as an issue in systems security and as a more theoretical issue in the study of classical information security and information flow theory. Several systems (Montagu, Pierce and Pollack 2013) apply dynamic taint tracking as a defensive measure to forbid dangerous interactions between untrusted input data and the system.

In taint tracking systems there are two categories of propagation (flows): explicit and implicit. Explicit flows are updates to the application taint state that are not predicated on some conditional. Implicit flows may be predicated on one or more conditionals. An implicit flow arises in a program when instead of a variable being directly assigned; the variable is assigned under the influence of a tainted variable. For example, we have two programs where T is a tainted variable and L is an untainted variable:

Figure 1: Explicit taint propagation

```
L := T;
```

Figure 2: Implicit taint propagation

```
if (T) {
   L := 0;
}
```

Figure 1 illustrates an explicit flow, while Figure 2 illustrates an implicit flow. In the explicit flow, the assignment is unconditional and the flow is straightforward. In the implicit flow, the assignment is conditional on the tainted variable.

When designing a taint tracking system, special care has to be taken when handling implicit flows. Systems that ignore implicit flows -- under-tainting -- can mischaracterize traces as being free from attacker influence. Additionally, tracking implicit flows can lead to over-approximation -- over-tainting, for example:

Figure 3: Example of over-approximation via implicit taint propagation

```
if (T) {
   L := 0;
   V := 1;
} else {
   L := 1;
   V := 1;
}
```

A system that tracked implicit flows with tainting could characterize both L and V as tainted, when the value of V is not affected by the conditional.

Some theoretical work (Russo and Sabelfeld 2010) addresses the implicit flow problem. The research shows that neither a wholly static nor a wholly dynamic taint tracking system can be both sound and complete. An ideal system would be a "hybrid" that utilizes both static and dynamic analyses. For the example in Figure 3, this system would consider both sides of the conditional and discover that the value of V did not vary and characterize V as untainted.

Taint tracking answers the question "Has value X been influenced by input data?" but doesn't quantify the level of influence. There are several methods of quantifying influence, such as channel capacity.

Channel capacity, as defined by McCamant (Newsome, McCamant and Song 2009), provides a metric defining how much input flows to an output. The channel capacity is calculated from the minimum-slice of a dataflow-graph where the entry vertex is the input and the exit vertex is the output. The intermediate vertices are created from the instructions in the program interacting with tainted data. An example from McCamants work is below, as a figure.

Figure 4: Channel capacity

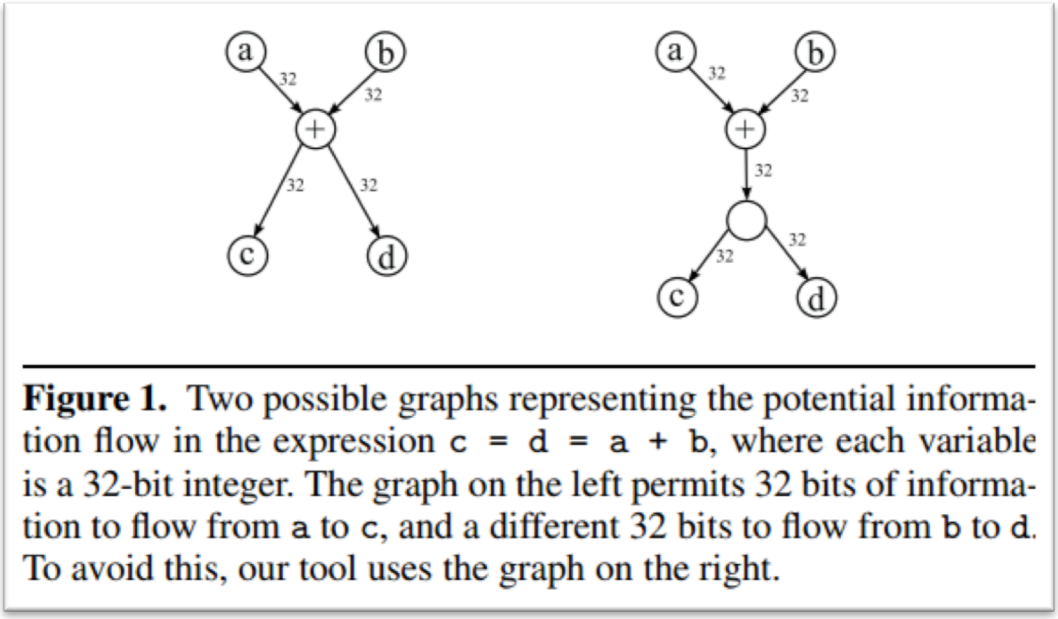

Figure 1. Two possible graphs representing the potential information flow in the expression `c = d = a + b`, where each variable is a 32-bit integer. The graph on the left permits 32 bits of information to flow from `a` to `c`, and a different 32 bits to flow from `b` to `d`. To avoid this, our tool uses the graph on the right.

Tracking channel capacity is more precise than taint alone. The value of a tainted variable is a single bit. Channel capacity provides a measure of how many bits in a variable the attacker controls. Applying taint tracking and channel capacity give rise to a natural characterization of a crash's severity. For example, suppose our fuzzing run has produced a crashing input with the following crashing program state:

Figure 5: Example crashing state

```
(ee8.c38): Access violation - code c0000005 (!!! second chance !!!)
rax=000007fffffde000 rbx=0000000000000000 rcx=000000000011d538
rdx=0000000000000030 rsi=0000000000000000 rdi=0000000000000000
rip=0000000076d311f1 rsp=000000000011d4e0 rbp=00000000ff653460
 r8=0000000076b49500  r9=0000000076b49500 r10=0000000000000000
r11=0000000000000344 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000

00000000`76d311df mov    r8d,dword ptr [rsp+2Ch]
00000000`76d311ed mov    r9,qword ptr [rax+58h]
00000000`76d311f1 call   qword ptr [r9+r8*8]      ds:00000004`2c593d00=?????????
```

In this example, the application crashed when trying to do an array lookup. It's not immediately clear what the cause was or how much influence an attack may have. Microsoft released a tool called !exploitable (Microsoft Security Engineering Center (MSEC) Security Science Team n.d.) which measures worst-case exploitability. !exploitable classifies our example crash as exploitable; however, it doesn't have a metric of measuring influence. If this crash was triggered with only a single bit of influence over the r9 register, that's much less likely to be exploitable than a scenario where an attacker has 64 bits of influence over r8 or r9. A crash is of extreme concern if an attacker can entirely influence the program execution. These crashes can sometimes turn out to not be exploitable due to the presence of ASLR and DEP in the program, however that is not guaranteed due to the possibility of an information leak allowing the PC to be set to the location of a ROP gadget.

1.1.2 Taint Tracking for Input Minimization

Taint tracking has obvious benefits when trying to identify attacker influence, but it can also be an incredibly useful tool to identify the minimum test case needed to exercise a vulnerable condition. Input minimization is an important problem when performing large scale fuzzing (Regehr, et al. 2012). Fuzzing frequently produces crashing test cases that are far larger than the minimum case. A crash in a C compiler could be 50,000 bytes, where the minimum case was only 200 bytes.

Input taint tracking can help answer questions about which sections of the input have relevance to the crash. It also helps during fuzzing by characterizing which parts of a file must be permuted to reach different regions of a program and which parts of the file are not considered by the program. This helps with manual root cause analysis as there is less information for a human analyst to consider.

Input taint tracking also has a clear applications to white-box fuzzing systems as well. Existing white-box fuzzing systems (Zalewski n.d.) use instruction coverage and stack-trace hashing techniques to measure progress. Subsequent fuzzing runs are then directed towards new code areas with this collected information. These systems could be extended by considering input influence along paths taken during fuzzing, encouraging the exploration of program states where input has more influence over the execution of the program.

1.1.3 Trace Analysis

Trace analysis can also help with identification of similar fundamental causes. Frequently, stack trace comparisons can be used to differentiate errors. In practice two crashing conditions that have a similar stack traces are frequently results of the same fundamental flaw. However, crashing conditions with different stack traces may still represent the same flaw. This is usually a function of the distance between the error and the manifestation. The distance is frequently constant for spatial memory errors, but can vary in temporal memory errors, also commonly known as "use-after-free" or UAF vulnerabilities.

Errors of this type can be binned correctly by considering the trace, the quantification of attacker influence, and common program points reached between the two traces. This could allow a triaging system for a fuzzer producing UAF crashes to recognize that two crashes with different call stacks actually have the same root cause.

Traces can be used to identify and characterize crashes relating to memory corruption, but could be used to identify and quantify other classes of vulnerabilities. While memory corruption leading to code execution is an especially pernicious class of vulnerability present in low-level code, similar issues can exist with equal or greater severity. An example of this is the Heartbleed vulnerability would allow an attacker to leak extremely sensitive data out of an application with high reliability.

1.1.4 Information Flow Theory

Information flow theory, an extension of taint tracking, can be used for fuzzing and crash triage. In information flow theory, data in the system can be given an integrity label based on the data's sensitivity. High integrity data is sensitive data in the system, while low integrity data is data introduced from outside the system. An analysis on traces can then search for violations of noninterference -- when low integrity data interferes with high integrity data in a manner that can be observed by an attacker. This interference and subsequent observation could result in the compromise of the data by the attacker. Information flow theory is directly applicable to finding and mitigating side-channel attacks in cryptosystems (Molnar, et al. 2005).

1.1.5 Symbolic Execution and Constraint Solving

Symbolic execution is a static analysis technique to explore the behavior of a program. Contrasted with concrete execution, symbolic execution uses symbolic values instead of concrete values (e.g. y = x + 3 vs. y = 4 + 3). During a symbolic execution, the system maintains unique states for each path with associated symbolic values and constraints on symbolic values. This allows a symbolic execution to take all feasible paths in a program. To determine the feasibility of states, symbolic executors frequently use formal logic and theorem proving software to determine if a set of symbolic values and their constraints create a contradiction. Most symbolic execution systems use satisfiability (SAT or SMT) solvers to prove contradictions.

Symbolic execution is useful in software testing because it produces fewer false positives compared to other static analysis techniques. If a symbolic execution system identifies a bug, it also produces a path through the program that leads to the bug and inputs that trigger the bug (Kil Cha, et al. 2012).

Symbolic execution as a technique is theoretically guaranteed to discover all errors in a program; however, there are no guarantees on the running time taken to do so. In practice this is the primary issue with symbolic execution. It is possible to make symbolic executors more useful by incorporating heuristics and more advanced scheduling algorithms to choose states to explore (Ma, et al. 2011).

Symbolic execution can be combined with concrete execution known as concolic execution -- combined concrete and symbolic execution. The concrete execution of the program provides an initial state for further symbolic execution. Concolic execution is interesting because generating concrete state is more efficient and it does not compromise the soundness of the symbolic execution.

Concolic execution systems can also act on traces generated during fuzzing with taint tracking information. Such a system could treat the tainted values as symbolic and use SMT constraint solving to symbolically explore how input affects the execution of the program. These systems have been explored before to enhance the effectiveness of fuzz testing (Campana n.d.). The fuzzer can use the information provided by the trace analysis and constraint solving to adjust the values provided as input to achieve greater code coverage.

We have experience with several real-world fuzzing systems that incorporate non-quantitative taint tracking for analysis and triage. When these systems are used to analyze large applications, such as web browsers, they produce an over-abundance of tainted data. To address this over-abundance, we believe insertion of abstraction boundaries (Schwartz 2014) around data processing will better focus vulnerability triage. Abstraction boundaries are boundaries where flows and constraints can be summarized over well-defined and well-understood libraries and transformations that would otherwise complicate analysis, such as lossless compression algorithms and localization conversions.

Abstraction boundaries are most effective in large software projects that accept many types of input data. For example, web browsers accept dozens of text encodings and file formats. For each text encoding, the web browser must convert the input to its native representation before it can act on the information. Each layer of translation propagates taint until the analysis is not worthwhile. Instead of considering taint as propagating from network input, we can consider taint as propagating from an API boundary where that boundary is related to the processing of input. This provides a tighter scope to the progression of taint through the program.

We're familiar with several existing systems that incorporate taint tracking, symbolic execution and constraint solving to aid in the fuzzing process such as AFL, moflow, fuzzgrind, and TEMU. We plan on utilizing each during our investigation into the ideal combination of analyses.

2. PHASE I TECHNICAL OBJECTIVES

The overall technical objective of the Automated Exploitability Reasoning solicitation is to develop a system that can be used to automatically measure and triage the exploitability of crashes utilizing program analysis techniques such as dynamic taint tracking and symbolic execution. This proposal addresses the initial phase of the program, which focuses on the feasibility of each analysis for such a system.

The phase 1 technical objectives are:

- Evaluate techniques such as dynamic taint analysis, symbolic execution, constraint solving, and concolic execution as a metric to evaluate the severity of software vulnerabilities.
- Develop an exploitability metric for software vulnerabilities utilizing the stated techniques.
- Develop recommendations for the development of a system that will automatically triage the exploitability of software vulnerabilities discovered via fuzzing.

Additionally, this proposal addresses an optional effort (Phase 1 Option) following the conclusion of Phase 1. The objective of the option is to take the recommendations from the initial phase and design a system to be implemented as part of subsequent work on the subject.

3. PHASE I - WORK PLAN

For the initial phase, we propose researching and evaluating dynamic taint analysis, symbolic execution, constraint solving, and concolic execution as a metric to evaluate the severity of software vulnerabilities discovered through fuzzing.

To better classify the severity of the software vulnerabilities, we will develop an exploitability taxonomy. Vulnerability classification in the taxonomy will be determined by quantitative influence as determined by the specific analyses being researched.

At the completion of the initial phase, we will compile a final report that details the applicability of each analysis technique as applied to determining the exploitability of software vulnerabilities and provide recommendations for such a system.

Additionally, we propose an optional effort (Phase 1 Option) following the conclusion of the initial phase. The option will take the recommendations from the initial phase and design a system that utilizes the research to be implemented as part of continued work on the subject.

3.1 Survey of techniques

To begin, we will review relevant literature from both academic research and industry. We will produce a report summarizing the literature we discovered. For this report, we will identify existing software systems that can be studied and evaluated. It's important that the systems can be applied to real-life problems; as such we will produce some benchmark test cases and evaluate those software systems on the test cases.

3.2 Collect a corpus for testing

Identification of a corpus of previous real-world vulnerabilities is vital to develop accurate recommendations. We will collect sample vulnerabilities from the CERT and NIST vulnerability databases. The vulnerabilities will be selected considering several criteria:

- Application complexity
- Vulnerability Impact
- MITRE's Common Weakness Enumeration (CWE) category and severity score (MITRE n.d.)

Both severe and minor vulnerability classes will be selected. Additionally, we will focus on vulnerabilities that can be found through the use of random and directed fuzzing. This corpus will be used to better gauge and quantify our system recommendations.

3.3 Exploitability Taxonomy

For accurate classification of vulnerability severity, we will develop an exploitability taxonomy. Several taxonomies exist and will be evaluated for their applicability. We expect to take an existing taxonomy and adapt it to our specific program uses, except with considerations made for the additional information provided by the analysis techniques we evaluated. Additionally, any taxonomy developed must be applicable automatically by the system to be developed in subsequent

phases. Once a taxonomy has been developed, we will utilize the experience of our exploitation experts to classify each corpus example and rank their severity.

3.4 Recommendations for system

Following the initial research into each analysis technique, we will quantify each in terms of performance required for accurate analysis, applicability to the problem, and difficulty of implementation. It's important to collect useful data in a reasonable amount of time and computing resources. For example, it shouldn't take a day to properly triage each crashing test case. Additionally, some analysis techniques, like symbolic execution, are incredibly complex and have huge memory and CPU requirements. For an accurate evaluation, we will utilize existing implementations when available to measure memory usage, CPU usage, and time elapsed to run each analysis technique. The collected data along with any recommendations will be compiled into a report that details the applicability of each analysis technique.

3.5 Final Report

At the completion of phase 1, we will provide a final report. The final report will contain all the system recommendations, the exploitability taxonomy, and the example corpus.

3.6 Phase 1 Option

In addition to phase 1, we propose an option phase directly following the completion of the initial phase. For the option phase, we will take the recommendations developed as a part of phase 1 and perform initial design of the system to be develop as part of subsequent phases.

3.7 Reporting

During the execution of the program we will provide status reports twice each month and full progress reports each month. At the completion of the project, we will provide a comprehensive final report.

4. STATEMENT OF WORK

Phase 1:

- The contractor will evaluate techniques such as dynamic taint analysis, symbolic execution, constraint solving, and concolic execution as a metric to evaluate the severity of software vulnerabilities discovered through fuzzing.
- The contractor will develop an exploitability taxonomy to categorize the severity of software vulnerabilities to enable better triage.
- The contractor will develop recommendations for which evaluation strategies best fit with certain classes of software.
- The contractor will provide status reports twice each month and full progress reports each month.
- The contractor will prepare a final report that details the applicability of each analysis technique as applied to determining software vulnerability exploitability.

Phase 1 Option:

- The contractor will take the recommendations from phase 1 and design a system to be developed as part of phase 2.
- The contractor will provide status reports twice each month and full progress reports each month

5. SCHEDULE

	Task Name	Work	Duration	Start	Finish	Predecessors
1	Program Start	0 hrs	0 days	Mon 6/22/15	Mon 6/22/15	
2	Phase 1 End	0 hrs	0 days	Fri 12/4/15	Fri 12/4/15	1SS+6 mons
3	Phase 1 Option Start	0 hrs	0 days	Fri 12/4/15	Fri 12/4/15	2
4	Phase 1 Option End	0 hrs	0 days	Fri 3/25/16	Fri 3/25/16	3SS+4 mons
5	**Phase 1**	**636 hrs**	**115 days**	**Mon 6/22/15**	**Fri 11/27/15**	**1**
6	**Survey of techniques**	**260 hrs**	**55 days**	**Mon 6/22/15**	**Fri 9/4/15**	
7	Review relevant academic and industry literature	60 hrs	2 wks	Mon 6/22/15	Fri 7/3/15	
8	Literature summary report	60 hrs	3 wks	Mon 7/6/15	Fri 7/24/15	7
9	Find examples of existing software systems	60 hrs	2 wks	Mon 7/27/15	Fri 8/7/15	8
10	Evaluate existing software systems	80 hrs	1 mon	Mon 8/10/15	Fri 9/4/15	9
11	**Collect a corpus for testing**	**76 hrs**	**10 days**	**Mon 6/22/15**	**Fri 7/3/15**	
12	**Identify a set of applications:**	**76 hrs**	**10 days**	**Mon 6/22/15**	**Fri 7/3/15**	
13	"Real-world" (web browsers, virtual machines, etc)	16 hrs	2 days	Mon 6/22/15	Tue 6/23/15	
14	Has bugs that can be found via fuzzing	30 hrs	1 wk	Mon 6/22/15	Fri 6/26/15	
15	Has some way to fuzz for the bugs	30 hrs	1 wk	Mon 6/29/15	Fri 7/3/15	14
16	**Exploitability Taxonomy**	**100 hrs**	**25 days**	**Mon 9/7/15**	**Fri 10/9/15**	**6,11**
17	Evaluate existing definitions	20 hrs	1 wk	Mon 9/7/15	Fri 9/11/15	
18	Classify examples into each of those definitions	20 hrs	1 wk	Mon 9/14/15	Fri 9/18/15	17
19	Rank the severity of each definition	20 hrs	1 wk	Mon 9/21/15	Fri 9/25/15	18
20	Rank each of the examples from the corpus by the definition	40 hrs	2 wks	Mon 9/28/15	Fri 10/9/15	19
21	**Recommendations for system**	**140 hrs**	**25 days**	**Mon 10/12/15**	**Fri 11/13/15**	**6,11,16**
22	Quantify each analysis in terms of performance and applicability	40 hrs	2 wks	Mon 10/12/15	Fri 10/23/15	
23	Quantify each analysis in terms of difficulty of implementation	40 hrs	2 wks	Mon 10/12/15	Fri 10/23/15	
24	Create a recommendation report for each analysis	60 hrs	3 wks	Mon 10/26/15	Fri 11/13/15	22,23
25	Final Report	60 hrs	2 wks	Mon 11/16/15	Fri 11/27/15	21
26	**Phase 1 Option**	**160 hrs**	**20 days**	**Mon 12/7/15**	**Fri 1/1/16**	**5,3SS**
27	Design Phase 2 system	160 hrs	1 mon	Mon 12/7/15	Fri 1/1/16	

6. RELATED WORK

Trail of Bits is currently performing on or has completed several projects in closely related areas.

6.1 DARPA Cyber Grand Challenge

We are currently competing as one of 7 funded teams in DARPA's Cyber Grand Challenge. As part of the competition, we're developing static and dynamic analysis tools to automatically detect, mitigate, and exploit software vulnerabilities.

6.2 Another DARPA program - mcsema

On another DARPA program, we developed and open-sourced mcsema, a framework that performs static translation of x86 binaries to the LLVM intermediate representation. mcsema allows us to leverage LLVM's comprehensive analysis platform on binary code for software transformation, vulnerability discovery, software optimization, and other uses. We continue to work on mcsema on several other pursuits. This framework has been adapted and used by other DARPA programs. The code is available at https://github.com/trailofbits/mcsema.

6.3 DARPA Cyber Fast Track - Code Reason

Code Reason is a research project to build a framework for symbolic reasoning about code and semantic discovery of code snippets that can be utilized in return-oriented-programming (ROP) attacks.

6.4 DARPA Cyber Fast Track - PointsTo

PointsTo is a research project that is intended to find object life-cycle (use-after-free, use-after-return) vulnerabilities in large software projects such as web browsers and web servers.

6.5 Assured Exploitation Training

The Assured Exploitation class provides students with a hands-on exploit development on modern platforms that employ exploit mitigations such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Guard Stack (GS). As part of the class assignments, students develop an exploit that defeats DEP through return-oriented-programming (ROP) and defeats ASLR by utilizing heap-feng-shui to craft an information-disclosure.

7. RELATIONSHIP WITH FUTURE RESEARCH OR RESEARCH AND DEVELOPMENT

Researching program analysis tools and techniques as well as the development of an exploitability taxonomy will be immediately useful to anyone looking to determine the severity of potential security vulnerabilities. The results of Phase 1 will directly feed into the design of an exploitability ranking system that will be developed as part of Phase 2.

The analysis and review performed during Phase 1 will create a roadmap of current techniques that use program analysis to reason about potential vulnerabilities. Phase 1 will also create a catalog of available tools and test cases for those tools. The identified strengths of analysis tools can be incorporated directly into a follow on system, and the identified weaknesses serve as a natural place to begin our own software development.

An exploitability taxonomy will be useful when communicating the progress and outcomes of fuzzing campaigns. This taxonomy will clearly show the severity of issues identified by fuzzing campaigns, helping prioritize resources to fix the most severe issues first.

8. COMMERCIALIZATION STRATEGY

Automated software vulnerability analysis is a growing commercial industry. Current approaches lack the ability to appropriately score exploitability of discovered vulnerabilities. By providing prioritized rankings of exploitability in closed-source software running in the enterprise, commercial organizations can now understand risks in previously opaque software. As such, there are three immediate markets for Sienna Locomotive,

a) a supplement for a software development team's quality assurance process,

b) an independent software security scoring and exploitability risk analysis metric available to software buyers and sellers, and
c) an independent tool for vulnerability assessment service providers to assist during assessments.

The successful application of automated vulnerability scoring will significantly increase the productivity of security researchers, software reverse engineers, and incident response professionals across the industry.

9. KEY PERSONNEL

Nicholas DePetrillo, Principal Investigator

Nick is a Principal Security Researcher at Trail of Bits where he focuses on mobile security and program management. At Trail of Bits, Nick manages all Government contracts as both a technical and administrative manager. Nick brings more than ten years of security research experience to Trail of Bits, most notably in the area of mobile security. Nick is widely regarded as one of the industry's foremost experts in the field, attracting considerable attention for his discovery of significant security flaws that impact the privacy of millions of smartphone users and wireless network customers. He has worked on research throughout the entire mobile phone technology stack including cell phone network infrastructure, baseband radio security research and at the application and operating system level. Prior to joining Trail of Bits, Nick was an independent consultant specializing in mobile security research services. He was also a Senior Security Researcher at Harris Corporation, focusing on mobile and wireless platforms. Nick is a frequent lecturer, and has presented his work at BlackHat and other security conferences around the world.

Relevant Experience

- Designed new tools to create a repeatable process for conducting future vulnerability research and exploit development.
- Developed firmware to implement new techniques on cellular collection equipment.
- Performed blackbox red-team testing and assessment to simulate an attacker who steals equipment and attempts a rapid reverse engineer to determine origin.
- Developed reverse engineering tools for the HEXAGON architecture used in Qualcomm baseband radios.
- Led several technical programs focused on exploit and vulnerability analysis and mobile security.
- Led hardware reverse engineering team for Harris' Wireless Products Group on unusual cellular collection and mobile products discovered in the wild.
- Performed initial industry hardware and software vulnerability research info the first generation of Smart Grid systems.
- Created and managed an Open Source software lab called "Aruba Labs" to maintain and foster the development of Open Source software and projects on Aruba Networks hardware.

- Developed a Linux based IDS appliance for "plug-and-play" use at various sites and network layouts.

Nick holds an active DoD Top Secret security clearance with an SSBI and a counter-intelligence polygraph.

9.1 Additional Personnel

Andrew Ruef and Ryan Stortz will serve in non-key subject matter expert roles.

Andrew Ruef, Program Analysis Expert

Andrew Ruef is a Senior Systems Engineer at Trail of Bits where he conducts information security analysis. Andrew has more than a decade of industry experience in computer network security and software engineering, working on various projects including reverse-engineering of malware, analysis of computer network traffic for security purposes, system administration, and development of secure software products. Andrew has spoken at a seminar on binary analysis and given many talks on reverse engineering and operating system internals to professional groups in the DC area. Andrew also co-presented the mcsema framework at ReCon 2014 and was invited to speak at Microsoft Bluehat 2014. Andrew was the primary investigator on a DARPA Cyber Fast Track grant researching static program analysis. Andrew has also co-written articles about software security and performed research on NSF grants in collaboration with senior researchers at the University of Maryland, where he is currently pursuing his PhD in computer science. Andrew consults as a subject matter expert in the field of binary program analysis.

Relevant Experience

- Performed research and implemented techniques in static software analysis.
- Developed a framework that leverages symbolic execution to reason about snippets of code with the goal of discovering return-oriented-programming gadgets.
- Developed a framework, mcsema, available open-source at https://github.com/trailofbits/mcsema that translates native x86 machine code into the LLVM intermediate representation to enable scalable program analyses built on the LLVM compiler framework.
- Co-inventor for a framework that classifies malicious software based on statically discernable behavior, described in patent US8533836.
- Developed a malware sandbox utilizing the PIN dynamic-binary-translation framework to capture system calls, trace code execution, and record network activity.
- Conducted research into low-level Windows operating system security features and architected new capabilities and components in C, C++ and hand-coded assembly.
- Developed a 7-day curriculum covering Windows driver development and operating system internals from a CNO perspective. Served as lead instructor, presented lectures and administered student labs.

Andrew holds an active DoD Secret security clearance.

Ryan Stortz, Exploitation and Fuzzing Expert

Ryan is a Senior Security Researcher at Trail of Bits. At Trail of Bits, Ryan focuses on reverse engineering and software vulnerability research. Ryan is currently conducting several commercial application audits as well as providing his mobile reverse engineering expertise to the MAST Obfuscating compiler project at Trail of Bits. Prior to Trail of Bits, Ryan was a Senior Security Researcher at Raytheon SI Government Solutions (SIGOVS) where he focused on reverse engineering, vulnerability discovery, and exploit development on a variety of operating systems and architectures.

Relevant Experience

- Performed security research on desktop, mobile, and network appliance software to discover exploitable software vulnerabilities using static analysis, fuzzing, and concolic testing techniques.
- Leveraged software vulnerabilities to develop exploits for the purpose of remote code execution, sandbox escapes, local elevations of privilege, code signing bypasses, and information disclosures.
- Developed exploits that bypass operating system and compiler-introduced exploit mitigations such as NX/DEP, ASLR, SafeSEH, SEHOP, Code Signing, and Mandatory Access Controls.
- Implemented and utilized program analysis techniques such as taint propagation, code and branch coverage, symbolic execution, and satisfiability (SMT-Lib) solving to analyze software for the purpose of vulnerability discovery and reverse engineering.
- Reverse engineered proprietary software implementations back to high-level representations (C/C++).
- Developed CNO tools to enable covert communications and process, file, and socket hiding.
- Developed Windows, Linux, and Mac OS X kernel drivers and corresponding applications.
- Developed debuggers, fuzzers, and software testing frameworks for commodity and proprietary systems and protocols.
- Lead several small (5-10) teams of security researchers with weekly, monthly, and quarterly reporting.
- Lead and participated in dozens of contract pursuits with multi-year values ranging from $100k to $30M.
- Created training materials focusing on teaching reverse engineering, vulnerability discovery, and exploit development.
- Organizes Ghost in the Shellcode, a capture-the-flag that runs yearly in Washington, DC. Participates in several capture-the-flag competitions, such as DEFCON CTF, CODEGATE, and PlaidCTF.

Ryan holds an active DoD Top Secret security clearance with an SSBI and a counter-intelligence polygraph.

10. FACILITIES/EQUIPMENT

Trail of Bits is a New York-based corporation with an office in the Financial District of Manhattan. Our employees are split evenly between the main office and remote work from several areas of the United States. To facilitate secure communications, we have deployed a secure engineering network for the majority of our work that includes secure chat, video, and file sharing services. There is no connectivity between the engineering network and the public Internet. For remote employees, we utilize an encrypted IPSEC secure tunnel with dedicated hardware.

11. CONSULTANTS

Trail of Bits has the relevant experience required to successfully perform on Phase 1. We have no plans to include consultants for Phase 1.

12. PRIOR, CURRENT, OR PENDING SUPPORT

Trail of Bits has no prior, current, or pending support for a similar proposal. Currently this proposal is the only solicitation Trail of Bits is actively pursing.

13. COST PROPOSAL

See attached.

14. REFERENCES

Campana, Gabriel. *Fuzzgrind.* http://esec-lab.sogeti.com/pages/Fuzzgrind .

Kil Cha, Sang, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." *IEEE Symposium on Security and Privacy .* 2012.

Ma, Kin-Keung, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Directed Symbolic Execution." *Lecture Notes in Computer Science*, 2011: 95-111.

Microsoft Security Engineering Center (MSEC) Security Science Team . *!exploitable Crash Analyzer .* https://msecdbg.codeplex.com/ .

MITRE. *CWE - Common Weakness Enumeration.* http://cwe.mitre.org/.

Molnar, David, Matt Piotrowski, David Schultz, and David Wagner. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks." *In Cryptology*, 2005.

Montagu, Benoit, Benjamin C. Pierce, and Randy Pollack. "A Theory of Information-Flow Labels ." *IEEE Computer Security Foundations Symposium.* 2013.

Newsome, James, Stephen McCamant, and Dawn Song. "Measuring Channel Capacity to Distinguish Undue Influence ." 2009.

Regehr, John, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-Case Reduction for C Compiler Bugs." *PLDI.* 2012.

Russo, Alejandro, and Andrei Sabelfeld. "Dynamic vs. Static Flow-Sensitive Security Analysis." *IEEE Computer Security Foundations Symposium.* 2010.

Schwartz, Edward J. *Abstraction Recovery for Scalable Static Binary Analysis.* May 2014. http://users.ece.cmu.edu/~ejschwar/papers/arthesis14.pdf.

Zalewski, Michał. *American Fuzzy Lop (AFL).* http://lcamtuf.coredump.cx/afl/ .