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 fuzz testing, or 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 powerful tool for security analysis, but when used by inexperienced researchers poses a number of challenges: fuzzing can find the same crash many times over, causing more work for the researcher; the fuzzing method used by the researcher may not apply (e.g., bit flipping bytes when ASCII text is required), rendering all fuzzed input useless; fuzzing doesn't take required structure into account (e.g., CRC inside the input); and even when fuzzing does find a crash, it provides no information about the exploitability of that crash. When fuzzing produces more crashes than are feasible to deeply investigate, some form of crash triage must be applied. A researcher must examine the context of the crash and the behavior of the target and make a judgment as to the severity of the crash and the probability of exploitation. This skill can take a significant amount of time and experience to develop, and as the number of crashes increase, the need for faster and more efficient crash triage techniques becomes apparent.

The cost of learning, understanding, and staying abreast of the state of the art of effective fuzzing and effective crash triage is insurmountable for many organizations. While consulting and short term contracts with security researchers can fill some of the gap, those solutions are at best expensive and piecemeal, rendering them infeasible for smaller projects and small organizations. A standalone crash triage tool would provide some utility to these users, but would still incur a significant investment of time for setup, use, and interpretation of results. Trail of Bits proposes an automated fuzzing and crash triage service that will enable a low-effort inclusion of security awareness for organizations or projects that do not have access to the time or expertise required to develop or hire dedicated security experts.

1.1 PHASE 1 RESULTS

The Phase 1 SBIR project surveyed the state of the art for crash triage tools. A number of tools exist with various capabilities, but all are limited in some manner and require a great deal of experience to use correctly. Additionally, Trail of Bits proposed and built a prototype crash triage tool based on taint tracking that addresses some of the limitations in extant crash triage tools and presents an approach that balances soundness and scalability. The tool and the example use case presented demonstrated that while complete correctness regarding exploitability is nontrivially computationally expensive, approximations using quantified influence can provide a ranking mechanism for the severity of a vulnerability.

The lack of an effective crash triage tool demonstrates the gap between the state of the art in security research and the state of the art in general commercial development. Security researchers

are familiar with crash triage and have developed tools that aid in their approach to it, but no such tool exists that can assist a developer who is inexperienced regarding fuzzing and crash triage.

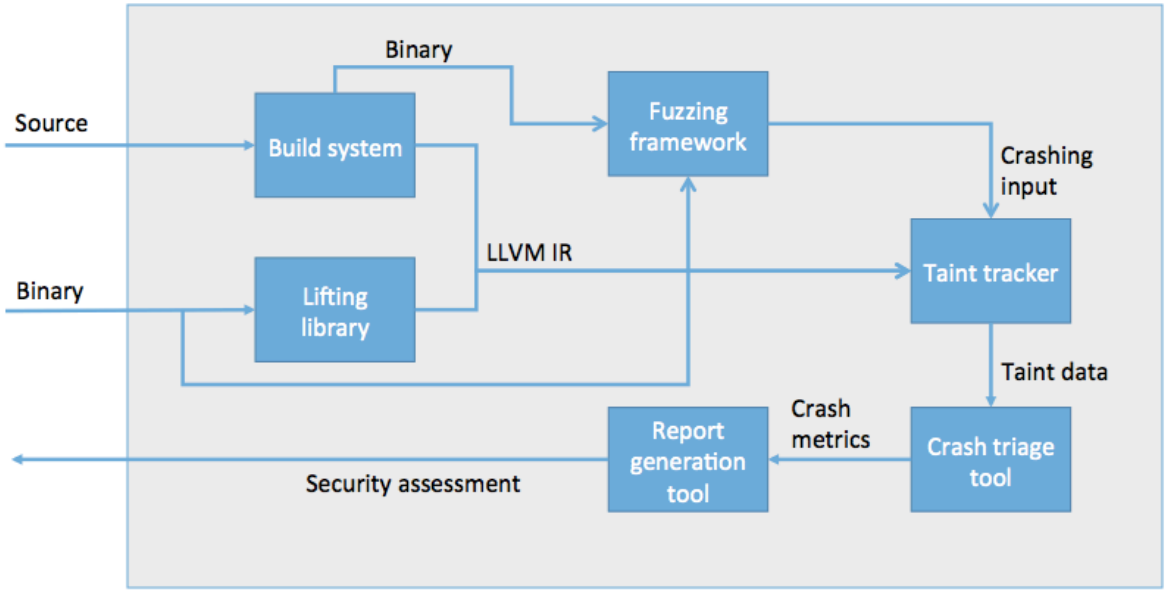
2. PHASE 2 TECHNICAL OBJECTIVES

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 second phase of the program, which focuses on the development and commercialization of such a system.

The Phase 2 technical objectives are:

- Design and implement a system to automatically triage crashes in an application
- Design and implement a system to automatically fuzz inputs for an application
- Design and implement a service that provides fuzzing and crash triage to users who may or may not have an understanding of best practices regarding fuzzing and crash triage

Trail of Bits will accomplish these objectives by building off existing internal tools, developing new tools, extending the prototype developed in Phase 1 of this project, and modifying existing open source tools. The resulting service will be composed of several parts: a lifting library, a build system, a taint tracking tool, a fuzzing framework, a report generation tool, a crash triage tool, and an interface of services and integrations. A high level overview of the interactions between these parts is shown in the figure below.



When a user submits their source code to the service the build system will check that the supplied source code is buildable and complete, then will perform the build to generate the target application. As a part of the build process a translation of the source into LLVM IR will be emitted.

Alternatively, the user may submit a compiled binary instead, which the lifting library will consume to create an LLVM IR translation of the binary. In either case, the result is a target application as well as a translated form of the application that supports analysis and interrogation. The target application is also consumed by the fuzzing framework, which generates and supplies new inputs to the target application in an effort to cause a crash. If a crash is detected the causative input as well as the translated application is provided to the taint tracking tool, which will follow the input through the execution of the translated application and generate data regarding the state of taint at the time of the application's crash. These data are fed to the report generation tool, which will summarize the data in a human readable manner and, when possible, provide potential avenues for amelioration of the crash or vulnerability. This report on discovered crashes, their severity, and possible mitigations is provided back to the user.

At the completion of Phase 2 the fuzzing and crash triage service will be ready for use by commercial development organizations. The proposed service will function as a verification and testing tool for the commercial and governmental markets. It is envisioned that the emphasis on ease of use and integration into existing build environments will reduce barriers to adoption.

Task 1 - Lifting Library Feature Additions
This task will build on an existing tool to create a platform for interrogating and reasoning about an application and its behavior.

Task 2 - Develop Taint Tracking Framework
The objective for Task 2 is to build a system that will monitor the progress of tainted data through an application.

Task 3 - Develop New Project Processing Bot
Will examine extant build integration projects for ease of extension and adaptation to meet the needs of the platform. One will be selected and modified to suit the needs of the proposed service.

Task 4 - Develop Fuzzing Harness
The objective of Task 4 is to build a framework for providing input data to applications under test as well as generating new corrupted input data.

Task 5 - Develop Triage and Code Score Capabilities
This task will build off of Tasks 2, 3, and 4 as well as the research from Phase 1 to perform an analysis of the overall security and vulnerability of an application.

Task 6 - Develop Reporting Capabilities
Will be devoted to developing a tool to generate summaries of the data produced from the Task 5 tools.

Task 7 - Develop User Interface
Will create the website that will serve as the primary user interface for clients.

Task 8 - System monitoring and abuse prevention

Will implement checks and safeguards to prevent unintentional or malicious abuse of the provided services.

Task 9 - System testing

The objective of Task 9 is to create a set of use cases that the service shall support, then generate the requisite artifacts to test the functionality and completeness of these use cases.

Task 10 - Final report and documentation

Will produce the final report and other documentation. This documentation will specify the supported use cases and capabilities of the service as well as provide examples of usage and interaction.

3. PHASE 2 - WORK PLAN

The Phase 2 work is designed to use the prototype developed in Phase 1 as a base and add the features and framework necessary to provide a commercially viable service.

TASK 1. Remill Feature Additions

Remill is an open-source library developed by Trail of Bits for translation of native code into LLVM intermediate representation (IR). The LLVM IR that is produced is true to the machine semantics of the original code. This translation enables sophisticated manipulation and analysis of the original code's behavior using an industry standard framework and tools.

Machine code on the x86 or x86-64 platforms is very complex, with many instructions having side effects not explicitly stated in the instruction itself; in addition, the x86 and x86-64 instruction sets are vast in terms of number of instructions available and utilized. For these reasons, analysis on machine code itself is complex; it must account for the sheer number of possible instructions as well as keeping track of the side effects. To deal with this problem the proposed service will use Remill to lift target applications to the LLVM IR. The LLVM IR provides a well-defined language for representing the behavior of an application and is structured to allow robust analysis and manipulation. Because the LLVM IR is used as an intermediate representation by the industry-standard LLVM compiler tool chain and others, there are many existing analyses that operate on LLVM IR, and there is a framework in place for easily running an analysis or transformation pass over a set of LLVM bitcode.

Remill is a tool created by Trail of Bits to lift machine code to LLVM IR. It implements a separation of the translated code from the memory model via user-implementable intrinsics, which facilitates activities like observing all memory accesses as they occur. Remill also handles control flow information in such a way as to make it easier for an analysis to discern the particular type of control flow that is occurring. This additional information regarding memory accesses and control flow semantics makes it simple to instrument the translated code for specific purposes, such as tracking data as it flows through an application.

At the time of this writing Remill supports translation of both x86 and x86-64 instruction sets, as well as the instructions found in most modern x86-64 CPUs, such as SSE, AVX, and AVX-512. Rather than develop a new mechanism for extracting control flow information from a target application, Remill uses IDA Pro to generate control flow information, allowing Remill to support any application type that is supported by IDA Pro. Remill is implemented in C++, which permits straightforward development and extension without sacrificing performance.

TASK 1.1 Restructure Remill

1.1.1 Implement Missing Instructions

At the time of this writing, Remill is not complete in terms of instruction implementation. Some instructions will have to be implemented in Remill to support this project.

1.1.2 Transparency Support

This task will develop the mechanisms that permit translated code to interact with non-translated code. As translated code does not attempt to maintain state in the exact format that non-translated code requires, any time execution flows from non-translated code into translated code a stub must be executed first to set up the state that the translated code expects.

1.1.3 Runtime Support

The objective of this task is to develop the API and the support for it needed to allow Remill to function as an instrumentation platform. Remill does not perform analysis on its own, but instead will provide hooks for other libraries to interrogate the application at various points of execution.

1.1.4 Operating System Support

This task will focus on implementing the special case handling for operating system specific behavior inside Remill. As an example, signals are transparent to Remill, so a special handler for the sigaction system call will be developed.

1.1.5 Concurrency Support

This task will implement support for analysis of concurrent programs via the implementation of memory barrier intrinsics and atomic region intrinsics.

TASK 1.2 Fuzzing Framework Support

Additional work on Remill's API will provide the functionality and features needed to support the manipulation of a target application by a fuzzing framework.

TASK 1.3 Dataflow Analysis Support

This task builds off the investigation performed in task 2.2; the format of the data produced and made available by Remill to external libraries may need to be adjusted to match what dataflow analysis tools require.

TASK 2. Implement Taint Tracking Tool

The objective of this task is to develop TaintSan, a taint tracking tool. Taint tracking is the process of marking a specific slice of data in an application as tainted and monitoring how that slice is manipulated and interacted upon by the application. The severity of a vulnerability can be calculated by tracking tainted input through an application and evaluating the influence the input has on the application at the point of the crash. The prototype tool built in Phase 1 demonstrated the effectiveness of taint tracking as a mechanism for crash triage.

TaintSan will be implemented to operate on LLVM IR and as part of an LLVM build chain. This will enable TaintSan to be easily integrated into a build process and will be able to take advantage of the functionality provided by LLVM, allowing for better maintainability and extensibility. This integration, when coupled with Remill's lifting capability, will make analyzing large collections of binaries feasible.

TASK 2.1 Establish Scope

This task will be devoted to exploring the taint tracking requirements of the proposed service. A number of factors can affect taint tracking performance, and the tradeoffs between speed versus accuracy and precision must be evaluated. An overall design for how TaintSan will operate will be established in this subtask.

TASK 2.2 Investigate Dataflow Sanitizer

Dataflow Sanitizer is a tool in the Clang tool chain that provides a generic dynamic dataflow analysis framework, making it an ideal candidate to serve as the base for TaintSan. Currently under development, its feature set and capabilities will be evaluated against the needs of TaintSan.

TASK 2.3 Implement Shadow Memory for Taint Tracking of Memory and Registers

This task will involve implementation of a shadow memory system to store taint information for the application being tested.

TASK 2.4 Implement Taint Model and Taint Data Format

This task will implement the design decisions made 2.1 regarding the propagation of taint as well as the granularity and associated metadata.

TASK 2.5 Implement Taint Sources and Taint Sinks

This task will create the interface by which external tools can interact with the state of taint inside a targeted application. This will provide the hook by which a fuzzing harness can mark a slice of data as tainted as well as provide a hook by which the report generation tool can interrogate the state of taint in the system.

TASK 2.6 Query Data at Crash

This task will build off Task 2.5 and provide an interface for interrogating the state of taint at the time of a crash in the targeted application, including hooks for catching when a crash occurs.

TASK 3. Develop New Project Processing Bot

This task will produce integration tools for modern development environments, specifically git and GitHub. This integration will greatly reduce the cost and effort of adoption of the proposed service.

TASK 3.1 Investigate git interaction tools

This task will survey existing tools that automate interaction with git and GitHub to determine what capabilities already exist and which will need to be modified or built to suit the service's needs.

TASK 3.2 Survey open source build integration projects

Subsequent effort will examine the capabilities available in extant open source build integration tools in an effort to determine which, if any, provide the features and support needed for this service.

TASK 3.3 Project Validation

This subtask will focus on building a tool that will check that a submitted source tree is valid with respect to building, i.e., all dependencies are present, the build file is complete and in a supported format.

TASK 3.4 Automated Building

Upon a build tree's successful validation this task will actually perform the building step, producing the target application as well as a translation of the application's source into LLVM IR, suitable for analysis and interrogation.

TASK 4. Develop Fuzzing Harness

The objective of this task is to develop a harness for supplying corrupted input to the application under test as well as alerting TaintSan to the location of the supplied input in the application.

TASK 4.1 Implement Basic Mechanism for Supplying Input to Application

The first part of the objective will be to get a very basic harness in place that will support the simplest test scenarios developed in Task 9, which will enable system testing early in the development lifecycle.

TASK 4.2 Implement Mechanism for Supplying Test Inputs to Harness

This task will build on Task 4.1 and extend the capabilities of the fuzzing harness, allowing it to select from a corpus of pre-existing inputs supplied by the user. Users who have interest in specific inputs or in crashes resulting from specific inputs will be able to focus the service on only those cases.

TASK 4.3 Implement Mechanism for Delivering Generated Inputs Back to Client

This task will track generated inputs that cause new crashes or are otherwise of notable interest, store those inputs, and provide them to the user, allowing a user to run their own tests against these inputs.

TASK 5. Triage / Generate Code Score

The objective of Task 5 is to build upon Tasks 2, 3, and 4 to provide an analysis of a crashing application with respect to the exploitability of the crash or crashes.

TASK 5.1 Implement Metrics from Phase 1 Report

The prototype code from Phase 1 will be extended and integrated to derive data from TaintSan and Remill and produce a more comprehensive set of metrics and analysis.

TASK 5.2 Implement high-level security checks

As part of an integrated security analysis service, the proposed service will perform additional security evaluations on the application, e.g., checking to determine if modern compiler security flags are enabled.

TASK 6. Report Generation

TASK 6.1 Implement Basic Output with Metrics

Initial report generation will be straightforward, consisting of direct output from Remill, TaintSan, and the fuzzing framework. This provides a quick path to an early prototype, allowing for incremental improvements during the course of development.

TASK 6.2 Smarter Output with Formatting and Summary

Building off Task 6.1, subsequent effort will improve the output, formatting it for better readability, automatically generating long form human friendly summaries of the results, and, when available, potential avenues for either mitigation or exploitation of detected issues.

TASK 6.3 New Crash Detection

As part of a unified build process, it is expected that the automated fuzzing suite will be run on an application many times during its development lifecycle. By maintaining a database of known crashes, the suite will be able to detect when a new crash is discovered in an application, allowing for developers to rectify the faulting code earlier in development when refactoring is easier. For security researchers, new crash detection can illustrate the changes between versions of a targeted application, allowing them to focus their efforts directly against the new attack surface.

TASK 6.4 Modern notification integrations

The system will be able to notify users via various mechanisms that a new crash has been detected and a report is available for viewing. This will allow users to remain updated on the progress of their analyses without having to manually check status. Targets include email, GitHub notifications, and Slack (a popular chat service with many similar notification systems built in).

TASK 7. Web Interface

TASK 7.1 Core Website Infrastructure

This task will design and create the framework that will support the basic functionality of the web service. This will wrap an interface around the tools developed in Tasks 1 through 6, providing a simplified front end appropriate for invocation via a remote user.

TASK 7.2 Implement Single Pass Use Case

This task will implement a bare-minimum version of the proposed service. The core functionality of the service, i.e., automatic fuzzing and crash triage, can be exercised with a minimum of overhead, allowing for a faster bugfix cycle in the core functionality.

TASK 7.3 Add User Accounts

It is expected that users will want some of the benefits of individual user accounts on this web platform, so user account support will be added. Users will be able to log in to review past runs of the service against their submissions and inspect past results.

TASK 8. Testing

TASK 8.1 Create a Baseline Set of User Stories

This task will create a variety of scenarios and use cases that the proposed service shall support. These use cases will provide a baseline set of expected behavior that will drive testing and development.

TASK 8.2 Create Suite of Baseline Test Scenarios

This task will create the needed artifacts and environments to test the scenarios and use cases developed in Task 8.1. These will serve as gauges to measure the progress of the development of the proposed service as well as illustrating where issues arise.

TASK 9. Documentation / Final Report

User interface documentation, how-to guides, and lists of supported and future capabilities will be created in this task. Metrics that measure the effectiveness of the proposed service will be generated, and design documents will be corrected to reflect changes that may have occurred during implementation.

4. SCHEDULE

ID	Task Mode	Task Name	Work	Predecessors	Start	Finish
1		Program Start	0 hrs		Mon 11/7/1	Mon 11/7/1
2		**Development and Testing**	**4,280 hrs**		**Mon 11/7/1**	**Fri 3/2/18**
3		**Lifting Library**	**1,040 hrs**		**Mon 11/7/1**	**Fri 6/2/17**
4		**Restructuring**	**760 hrs**		**Mon 11/7/1**	**Fri 3/17/17**
5		Implement missing instructions	120 hrs		Mon 1/30/1	Fri 2/17/17
6		Transparency support	160 hrs		Mon 11/7/1	Fri 12/2/16
7		Runtime support	160 hrs		Mon 12/5/1	Fri 12/30/16
8		Operating System support	160 hrs	6,7	Mon 1/2/17	Fri 1/27/17
9		Concurrency support	160 hrs	7	Mon 2/20/1	Fri 3/17/17
10		Fuzzing framework support	120 hrs	4	Mon 5/15/1	Fri 6/2/17
11		Dataflow analysis support	160 hrs	4	Mon 3/20/1	Fri 4/14/17
12		**Traint Tracking Tool**	**800 hrs**		**Mon 11/7/1**	**Fri 4/14/17**
13		Establish scope and design	120 hrs		Mon 11/7/1	Fri 11/25/16
14		Investigate dataflow sanitizer	120 hrs	13	Mon 3/27/1	Fri 4/14/17
15		Implement shadow memory for taint tracking memory and registe	160 hrs	13	Mon 11/28/	Fri 12/23/16
16		Implement taint model and data format	160 hrs	13	Mon 2/27/1	Fri 3/24/17
17		Implement taint sources and sinks	120 hrs	15	Mon 12/26/	Fri 1/13/17
18		Implement data query at crash	120 hrs	17	Mon 1/16/1	Fri 2/3/17
19		**New Project processing bot**	**480 hrs**		**Mon 2/6/17**	**Fri 6/16/17**
20		Investigate git interaction tools	120 hrs		Mon 4/17/1	Fri 5/5/17
21		Test open source integration tools	120 hrs		Mon 2/6/17	Fri 2/24/17
22		Implement project validation	120 hrs	20	Mon 5/8/17	Fri 5/26/17
23		Implement project building	120 hrs	22	Mon 5/29/1	Fri 6/16/17
24		**Fuzzing Harness**	**480 hrs**	**3,19**	**Mon 6/19/1**	**Fri 9/8/17**
25		Implement basic input mechanism for application	200 hrs		Mon 6/19/1	Fri 7/21/17
26		Implement mechanism for supplying test inputs to harness	160 hrs		Mon 7/24/1	Fri 8/18/17
27		Implement mechanism for supplying generated inputs back to clie	120 hrs		Mon 8/21/1	Fri 9/8/17
28		**Triage / Code score**	**320 hrs**	**12**	**Mon 4/17/1**	**Fri 9/22/17**
29		Implement metrics from phase 1	160 hrs		Mon 6/5/17	Fri 9/22/17
30		Implement high level security checks	160 hrs		Mon 4/17/1	Fri 5/12/17
31		**Report Generation**	**520 hrs**	**28**	**Mon 9/25/1**	**Fri 12/22/17**
32		Implement basic textdump output	80 hrs		Mon 9/25/1	Fri 10/6/17
33		Add formatting and summary	120 hrs	32	Mon 10/9/1	Fri 10/27/17
34		New crash detection	160 hrs		Mon 11/27/	Fri 12/22/17
35		Slack / email integration	160 hrs	33	Mon 10/30/	Fri 11/24/17
36		**Web Interface**	**400 hrs**	**31**	**Mon 12/25/**	**Fri 3/2/18**
37		Establish core infrastructure	160 hrs		Mon 12/25/	Fri 1/19/18
38		Implement single session pass	120 hrs	37	Mon 1/22/1	Fri 2/9/18
39		Implement user accounts	120 hrs	38	Mon 2/12/1	Fri 3/2/18
40		**Testing**	**240 hrs**		**Mon 6/19/1**	**Fri 7/28/17**
41		Establish baseline user stories	120 hrs		Mon 6/19/1	Fri 7/7/17
42		Create baseline test suite	120 hrs	41	Mon 7/10/1	Fri 7/28/17
43		Final Report	80 hrs	12,19,24,28,	Mon 3/5/18	Fri 3/9/18

6. RELATED WORK

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

6.1 Zlib Security Audit

Trail of Bits is performing an automated security audit of zlib for the non-profit Mozilla Foundation, the maintainers of the Firefox web browser and Thunderbird email client. Zlib is an open source

compression library with a history of security vulnerabilities; Mozilla is interested in a security audit as the library is used in every product they ship. Trail of Bits is using the automated vulnerability discovery tools developed for the Cyber Grand Challenge to automatically audit zlib for security vulnerabilities, delivering confidence while reducing costs to a level acceptable by a non-profit.

6.2 Osquery

Facebook's osquery is an open source endpoint security tool that allows an organization to treat its infrastructure as a database. Using osquery, it is possible to issue SQL queries about organization state (e.g. list every process that is listening on port 445 across the entire network infrastructure). Osquery is used internally at Facebook and multiple other organizations to detect endpoint and network anomalies. Facebook has contracted Trail of Bits to port the osquery agent to Windows; currently osquery is Linux and Mac OS X only. All development is open source and continually committed to the main osquery tree. The current project state can be viewed at https://github.com/facebook/osquery.

6.3 DARPA Cyber Grand Challenge

We competed 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. These tools are now used for performing automated audits of commercial software.

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

Trail of Bits anticipates building a service that will provide automated building, lifting, fuzzing, crash triage, and report generation for users. The service will be compartmentalized in such a manner as to enable users to invoke only specific parts of the service, i.e., a user that already has inputs that cause crashes can invoke only the crash triage and report generation modules of the service.

The creation of this service would provide a strong foundation for further research and development efforts into crash triage techniques. The platform could also be extended to support research into automating vulnerability exploitation or mitigation.

8. COMMERCIALIZATION STRATEGY

Automated software vulnerability analysis is a growing commercial industry, as more and more IT professionals realize the importance of integrating security into the development cycle. 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 the magnitude of risk in previously opaque software and use this understanding to prioritize bug fixes appropriately. As such, there are three immediate markets for the proposed service:

a) an enhancement of 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.

As crash triage is currently a manual process that is both time consuming and relatively low value, augmenting with automation will significantly increase the productivity of security researchers, software reverse engineers, and incident response professionals across the industry. The inclusion of managed fuzzing and additional security checks will encourage greater access to the service, increasing the commercial viability. In keeping with the federal government policy of open source, Trail of Bits plans to open source the fuzzing framework and the taint tracking tool. Previous projects open sourced by Trail of Bits have been adopted by the security community and continue to see significant contributions and use by other teams.

There is not a strong commercial market for a standalone crash triage tool; it would be a specialized tool that would only be required by users that have fuzzed tested an application but have too many results to sift through manually or have insufficient security expertise to interpret the results of

their fuzz testing. Most users in these situations will either hire a security consultant, who will focus on more specialized fuzzing, or will forgo further security testing.

This state of affairs leaves an unaddressed segment of the market: users who desire security awareness of a product, but lack the time or cannot justify the expense of hiring a security consultant. By providing a low effort mechanism for security feedback at a price lower than hiring a security expert, Trail of Bits expects to address this audience.

Data on the size of this market is not directly available, but a correlation can be drawn between the proposed service and existing services that aid with development. Travis-CI (http://travis-ci.com) is a service that integrates with GitHub and provides feedback on test cases for an application under development; it operates on a subscription based fee schedule and has, from its website, “more than 250,000 users”. While not all users of Travis-CI will use a security assessment service, Trail of Bits estimates that 1% of the Travis-CI user base would find this service useful, yielding a reasonable target user base of 2,500 users.

Trail of Bits anticipates pricing this service at $1500 per user per month. With an average anticipated base of 2,500 users per year, this produces a target annual revenue of $3.75 million.

After development of this service, the effort to bring it to market is anticipated to be minimal. As the service will be hosted and operated by Trail of Bits, using commercial hosting companies to provide the necessary server space and network connectivity. Profit from existing contracts will be used to pay for this deployment until subscriptions provide enough revenue to account for the hosting costs of the service.

Trail of Bits has experience marketing to the security technology industry and does not anticipate incurring additional costs for marketing the service.

Trail of Bits offers security audits and current commercial rates are approximately $12000 per engineer per week, with small efforts usually requiring four to six weeks of engineer time, which equates to a minimum spend of at least $50,000. Depending on the results of the audit, mitigation of the discovered weaknesses may require new additional audits. These costs are in line with known costs from competitors in the security audit market. There does not, at this time, appear to be competition yet for automated fuzzing and crash triage as a service. By providing this service and pricing it below the cost of an audit by a dedicated security researcher, Trail of Bits presents a substantially cheaper alternative than competing security research firms.

Trail of Bits will report the number of subscriptions, the number of audits performed, the sales revenue, and the ongoing maintenance and developments costs for the service.

9. KEY PERSONNEL

Yan Ivnitskiy

Yan Ivnitskiy is a Principal Security Architect at Trail of Bits. Yan brings over 10 years of industry experience, working on compiler-based software analysis projects, hardware security experiments, and research prototyping. Prior to Trail of Bits, Yan discovered vulnerabilities in enterprise software and advised in remediation as a Senior Security Consultant at Matasano Security (Now part of NCC Group). Prior to Matasano, Yan was an Analyst at the National Security Agency. Mr. Ivnitskiy holds an MS and a BS degree in Computer Science from Polytechnic Institute of New York University.

Relevant Experience

- Performed security assessments of system software on Linux, Windows, and OS X-based targets.
- Developed dynamic binary analysis tools for the express purpose of vulnerability discovery.
- Developed a static source analysis prototype to aid analysts in understanding large code bases.
- Was lead engineer on an embedded firmware suite prototype to augment the security capabilities of mobile devices.
- Reverse-engineered program binaries and protocols for security assessments, including proprietary financial trading and SCADA industrial control systems.
- Co-authored a suite of Linux kernel modules for integrity verification of live systems.
- Developed and co-authored configuration guidance for the iOS and OS X operating systems.

Artem Dinaburg

Mr. Artem Dinaburg has over eleven years of extensive software engineering experience working in application software development, low-level software development, vulnerability research, reverse engineering, malicious software analysis, and program analysis. Mr. Dinaburg has spoken at academic and industry conferences such as ACM CCS, DEFCON, and Blackhat, and ReCON.

Mr. Dinaburg was the Principal Investigator for Trail of Bits' DARPA Cyber Grand Challenge (CGC) team. Under his leadership, the Trail of Bits CGC team developed Cyberdyne, a system to automatically identify and patch vulnerabilities in binary software. Currently Mr. Dinaburg is working on extending the capabilities of Cyberdyne and on other automated vulnerability discovery tools. Prior to joining Trail of Bits, Mr. Dinaburg worked at Raytheon on a wide range of projects ranging from program analysis to custom tool development for government clients.

Relevant Experience

- Re-architected a medium-detail radar simulator, written in C on Windows 3.1, to a modern object-oriented modular architecture using C++ and the g++ compiler
- Modified an existing Windows binary obfuscation tool to process modern PE files, such as adding support for certificates, relocations, and the delay load import table.

- Co-Developed McSema, an x86 to LLVM bitcode transformation tool. McSema is able to take existing windows binaries and transform them to LLVM to facilitate program analysis. McSema supports most of the x86 CPU, including integer, floating point, and SSE registers.

9.1 Additional Personnel

Ryan Stortz will serve in a non-key subject matter expert role.

Ryan Stortz, Exploitation and Fuzzing Expert

Ryan is a Principal 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 2. We have no plans to include consultants for Phase 2.

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/ .