Automated Exploit Chain Modeling and Analysis

Thomas Wahl[1], Nicholas White[1], Guang Jin[1], Sukarno Mertoguno[2], Kevin Stevens[2] and Froy Maldonado[2]

[1]Trusted Science and Technology, Inc., Rockville, MD, USA

[2]Georgia Institute of Technology, Atlanta,GA, USA

twahl@trustedst.com (corresponding author)

Abstract: We describe early-stage research and tool development efforts to formally model and analyze *exploit chains*. These are sequences of exploits carefully crafted by an attacker to achieve an elaborate end-goal, such as an escalation of privileges of the executing thread. In this work, we are taking a systematic approach to constructing *formal models* of exploit chains in the form of finite-state machines, which are then converted into constraint-based semantic representations or timed automata, in order to analyze chains against metrics such as effectiveness, ease of reproduction, and stability under system variations.

Keywords: Exploits, Exploit chains, Vulnerabilities, Formal modeling, Analysis

1. Introduction and Background

While the concept of security exploits has been the subject of extensive studies, exploit *chains* have enjoyed less attention in the research community; existing work tends to be ad-hoc. What makes such chains challenging to recognize and understand is their trademark behavior of crossing diverse layers of a complex computational system (such as a mobile phone), including the device firmware, driver code, operating system code (e.g., Android kernel routines), user-level code, and application code (e.g., web browsers).

1.1 Exploit Chains

These are sequences of exploits designed so that one exploit achieves a sub-goal of the chain (such as escaping a security sandbox or the execution of attacker-provided code) that serves as the gateway to the next exploit. The final exploit accomplishes the ultimate attacker goal, such as an escalation of privileges of the executing thread, or the exfiltration of some data meant to be inaccessible to standard users. Individual exploits are typically enabled by *vulnerabilities* in the underlying code, such as dereferencing a pointer to freed memory.

Consider the example of an exploit chain capable of remotely creating a root shell on a wide range of Android devices (Gong, 2020). The chain connects three exploits, based on three vulnerabilities, which target diverse software components and span multiple layers of an Android device:

- A memory overwrite access vulnerability (CVE-2019-5877) in a JavaScript engine;
- A use-after-free vulnerability (CVE-2019-5870) used to enable executing a shell system command; and
- A vulnerability in the Qualcomm GPU Linux kernel driver (CVE-2019-10567) used to switch into boot mode that results in a root shell, and reboot.

1.2 Formal Methods

This is an umbrella term for a collection of techniques that represent computational systems as *formal models*, such as labeled graph structures or algebraic constraints, and then use advanced graph search techniques or mathematical logic, respectively, to establish properties of these models. Their key strength is that, due to their algorithmic and logical rigor, they can provide guarantees of the correctness of the established model properties. A downside is that the model representations (i.e., graphs, constraint sets, logical formulas) are often far away from mainstream programming languages (requiring extensive abstraction), and that the analysis techniques are expensive and may require non-trivial human expertise to use effectively.

2. Modeling and Analysis of Exploit Chains: An Approach

Figure 1 shows an overview of our approach. Our project is not about detecting or root-causing exploit chains, but about modeling and analyzing known chains, for the (future) purposes of diversification and extension. We therefore start with an exploit chain given as executable code, as well as sources for the attack code, if available. We first apply *static and dynamic analyses* to extract a finite-state machine (FSM) model from the given description; we refer to this stage as *exploit chain ingestion*. The FSM as a model is generic in nature and lends itself to translation into different back-end analysis technologies: The figure shows the steps to extract either a

symbolic model, to be processed with a constraint solver, or a state-based model using timed automata, processed by the UPPAAL model checker (Bengtsson et al, 1996).

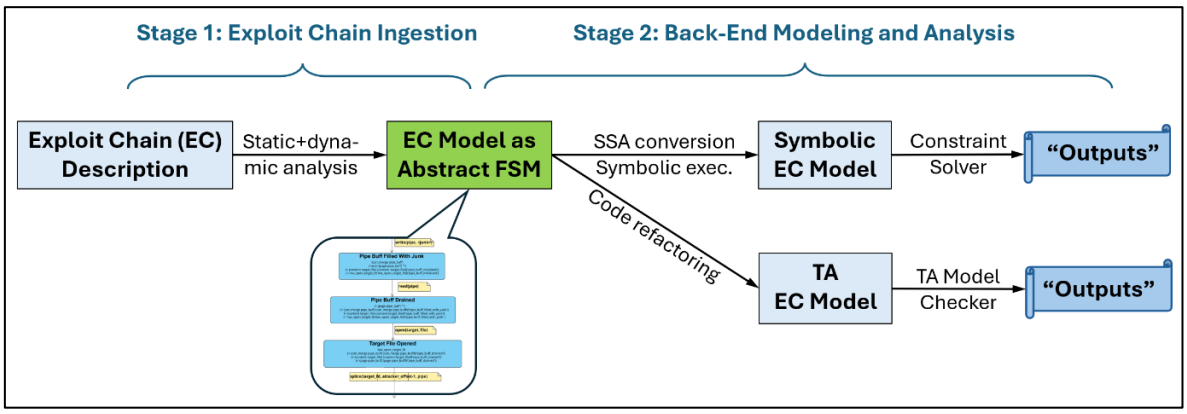

Figure 1: Overview of the two-stage exploit chain (EC) modelling and analysis process. An abstract FSM is constructed as intermediate representation. Obtaining a symbolic EC model requires converting the code into single static assignment (SSA) form, followed by symbolic execution

2.1 Exploit-Chain Modeling as a Finite-State Machine

The extraction of the FSM model employs both dynamic and static analysis techniques, at different stages of the ingestion algorithm, which we sketch in the following.

- **Determine a set of events that mark major milestones in the execution of the chain.** This step requires some high-level insights from a human expert into the exploit chain and is therefore done largely manually. Technically, the events are function calls in the various source code sections traversed by the chain. In addition, the events set includes memory allocation and deallocation statements, and thread spawns or joins.
- **Place log message instructions into the code at the event locations, and rebuild the chain code.** Challenges to automating this step include managing the various source code forms that exploit chains traverse, such as C/C++, Java, JavaScript, and shell scripts, all with different instrumentation requirements.
- **Execute the instrumented chain code, multiple times if necessary, until the chain has met its goal.** The execution generates a trace of the chain events previously identified as milestones. Tracing an execution of a full exploit chain targeting Android phones is very challenging: Such chains are bound to specific Android OS versions and other phone-specific software. We thus need either a mechanism to trace code running on a physical phone, or an Android OS emulator; we use both forms of tracing. The events serve as *states* of the FSM to be constructed.
- **Assign an execution semantics to the edges of the FSM.** This is accomplished by symbolically executing the code between two marker locations, defining the transition semantics via formulas that describe the change of state. While symbolic execution can be expensive, note that the trace dynamically constructed *focuses* the symbolic execution to a specific, thin slice of the overall code. The slice consists mostly of the path followed by the chain, but also preconditions checked on the way whose violation leads to termination of the attack. These checks are critical for assessing the robustness of the chain.

2.2 Exploit-Chain Analysis via Constraint Solving and Model Checking

The back-end analysis involves translating the FSM model into the input language of one of the available analysis tools, and run that tool. The table below compares the two tool pipelines that we support.

EC modeling language:	**Symbolic expressions**	**State transition system**
Analysis tools:	Constraint solvers: Z3, CVC4, …	(Explicit-state) Model checkers: UPPAAL, SPIN
Advantages compared to the other tool set:	Can analyze chain behavior under side conditions Can reverse-engineer inputs for a given effect	Good at handling multi-threaded scenarios Timed automata can track timing dependencies

An overview of the two pipelines is shown in Figure 2. The upper pipeline translates the FSM into the SMT ("Satisfiability modulo theories") language widely used by constraint solvers. The SMT model is then passed to a solver; we use Z3 (de Moura et al., 2008). The lower pipeline translates the FSM into the Timed Automaton (TA) input language of the UPPAAL model checker (Bengtsson et al., 1996). In both cases, the translation also takes attacker inputs and other configuration parameters as input, as well as a property specification. The property might express that the final state of the running exploit chain has given one of the participating threads root privileges, or that some content of a read-only database has been copied into a user (= attacker) variable.

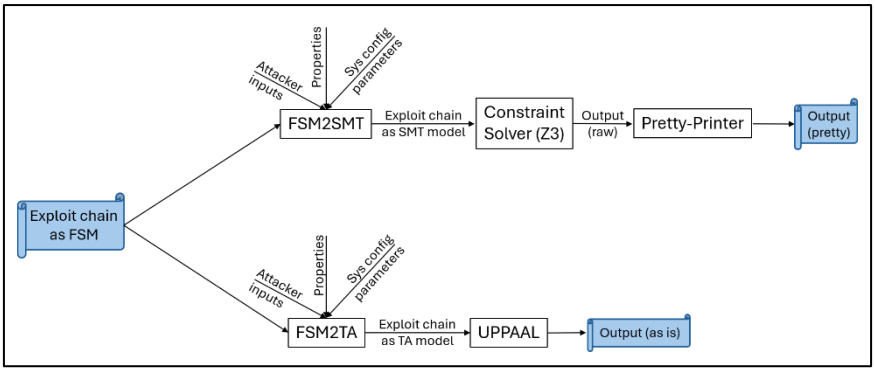

Figure 2: Two back-end analysis pipelines operating on top of the FSM

The decision which analysis pipeline to use is largely manual at this time but is of course heavily informed by the capabilities of the analysis tool. For example, an attack involving multiple threads will likely be processed using the UPPAAL-based pipeline.

2.3 Preliminary Results

We have so far created an initial implementation of the ingestion of a runnable exploit chain into an analysis model, and we have a decent implementation of the subsequent assessment of chain properties. We have used the implementation to analyze exploit chains at the kernel level, such as the infamous Dirty Pipe file-overwrite exploit (https://dirtypipe.cm4all.com), and Android exploits targeting mobile phones. We present preliminary results using the Dirty Pipe.

Property	Result	Interpretation
P1: E <> M0.end	Satisfied	We can reach the end state
P2: A [] M0.end => imply M0.main_1_ret_val == 0	Satisfied	Whenever the end state is reached, the Dirty Pipe has succeeded.
P3: Pr[<= 100] (<> M0.end)	P=[0.950056,1]	

The table above shows three Dirty Pipe properties P1..P3 in temporal-logic notation (as accepted by the UPPAAL model checker), the checker outcome, and an explanation. For example, P1 expresses that, along some path (E), machine M0 will eventually (<>) reach the end state. P3 asks: for paths up to 100 time units long, what is the probability of reaching the end state? In some cases it is necessary to introduce *ghost variables* into the model, solely used to express these properties. For example, the Dirty Pipe aims to overwrite a file that the attacker does not have write access to. Rather than modeling the complete Android filesystem, we model files as strings, embellished with a file access permission attribute (and others). The overwrite outcome of the Dirty Pipe is tested in the model by string comparison; if a change is observed, variable ret_val of function main_1 is set to 0 (property P2).

Property	States explored	CPU time (ms)	Virtual memory (kB)	Resident memory (kB)
P1	11441	56	154244	107607
P2	11442	61	154925	108135
P3	1647504	242	155141	108780

The table above quantifies the cost of analyzing the properties over the timed-automaton model, using UPPAAL. We see that, while the running times are very small, the memory requirements are sizable. However, this is in part due to a large start-up memory cost for state machine representation. We have run separate experiments,

using artificial examples, demonstrating that the memory requirements grow reasonably against growing model size.

3. Future Work

In addition to significant development work on the current preliminary implementation (such as the collaboration of the dynamic and static analyses for ingesting the exploit chains into an FSM), future plans include developing techniques for automating the repair of chains (if necessary), in the face of

- **incidental system changes:** generic kernel or software updates, and
- **system hardening:** patching of a vulnerability involved in the chain.

More ambitious yet is the goal of *forecasting* exploit chains: given a system and an anticipated software upgrade, what new opportunities for exploit chain creation does the upgraded system offer? Answers to such questions may furnish insights into both adversarial use of exploit chains, as well as defense against them. Both chain repair and forecasting likely offer several solutions per scenario, rather than a "unique best". We may consider Multi-Criteria Decision Making approaches under uncertainty to present the user with an informed choice.

Acknowledgements

This research was developed with funding from the US Defense Advanced Research Projects Agency (DARPA) under the "INGOTS: Intelligent Generation of Tools for Security" program.

Disclaimer: The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

AI Declaration: AI tools were not used in the creation of this paper.

Ethics Declaration: Ethical clearance was not required for this research.

References

de Moura, Leonardo Mendonca and Bjorner, Nikolaj (2008) "Z3: An Efficient SMT Solver", *Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS). DOI: https://doi.org/10.1007/978-3-540-78800-3_24

Gong, Guang (2020) "TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices", Black Hat USA.

Bengtsson, Johan and Larsen, Kim and Larsson, Fredrik and Pettersson, Paul and Yi, Wang (1996) "UPPAAL—a tool suite for automatic verification of real-time systems", DIMACS/SYCON workshop on Hybrid systems III: Verification and Control. DOI: https://doi.org/10.1007/BFb0020949