Exploitation as Code Reuse: On the Need of Formalization

Sergey Bratus, Anna Shubina

Abstract: This position paper discusses the need for modeling exploit computations and discusses possible formal approaches to it.

ACM CCS: Security and privacy → Formal methods and theory of security.

Keywords: Exploitation, modeling, weird machines.

1 Introduction

Exploitation of a vulnerability in software or hardware is making the system containing the vulnerability behave unexpectedly. Since *trustworthiness* of a machine or device is a matter of its expected behaviors, exploitation is *the* central phenomenon of computer security. Academic computer security advances by formal models of the computing phenomena it studies. Surprisingly, though, it still lacks a unified formal approach to describing exploitation.

One may argue that the concept of exploitation is so intuitive that it does not need formal definitions. Although we used to believe so in the past, we doubt it now. For example, is being exploitable a property of a particular bug or feature, or a whole-system property of the containing system? Our current terminology confuses these alternatives.

For a more substantive confusion, consider vastly divergent treatment of exploitation across different technological domains. Academic literature on exploitation of executable binaries overwhelmingly uses the device of Turing completeness as a criterion of viable contribution, whereas literature on exploitation of web application platforms almost never uses this device, focusing instead on the specific techniques used (SQLi, XSS, etc.) or specific kinds of information that can be stolen. This is despite the fact that exploitation in both domains is driven by crafted inputs and produces complex unexpected computations with unexpectedly meager means. This disparity is striking, and it persists across other target domains.

When closely related phenomena are treated so differently, their root causes are masked, and their analysis is muddled. Research effort, after all, goes where success is evident and likely to be recognized; our very view of the security domain may be skewed due to the lack of uniform criteria. For example, Turing completeness of exploit programming models is a reasonable proxy for the *composability* of their primitives—but is not equal to it. We may be under-studying practically important exploit models that are composable but not Turing complete.

In this position paper, we attempt to unify the different ideas regarding exploitation, and unpack the concepts possibly considered obvious—and thus overlooked.

2 Prior work

The understanding of exploitation as a programming discipline focusing on unexpected computation has a long history in the hacker community. Bratus et al. [8] give a sketch of how this discipline developed from leveraging specific cases of memory corruption to chaining up multiple primitives into powerful and generic programming models—up to Turing-completeness.

Exploits were understood both as proofs-by-construction of the unexpected computation's existence (a.k.a. vulnerability) and as programs in their own right. But if exploits are programs, and generalize into programming models, what machines do they run on?

Tautologically, exploits violate the expectations of the original programmer or designer of the vulnerable software, and hence their models of the target machine. These programmers or designers failed to see a richer machine embedded or emergent in the target. Exploits run on these richer machines, and are proof-by-construction that these richer machines exist.

The term "weird machines", informally coined in [7] and expanded by [9], attempted to capture this intuition, by

introducing the idea that an exploit's underlying execution model extended the programmer abstraction of the machine with additional "weird" states and transitions, arising, e.g., from memory corruptions or leaky abstractions. The underlying implementations or architectures made these states *representable* and *manipulable*. The manipulation (primarily, by means of crafted inputs) was the subject of exploit programming, of "*setting up, instantiating, and programming the weird machine.*" [9]

So formulated, the concept of a "weird machine" is intuitive, and was indeed instantly recognizable to many exploitation researchers. This intuition has been applied to a variety of programming models such as computations driven by ELF metadata [21], x86 memory descriptors [4], Linux kernel's signal-processing data structures [5], and Windows kernel's memory deduplication functionality [6], as well as others. Still, although productive, it lacked a formal definition.

In [24], Vanegue produced the first formal definition for the weird machine as a computational phenomenon in the context of Proof-Carrying Code (PCC). Dullien in [10] constructs an architectural model demonstrating the emergence of unexpected computation on a formally defined platform. Vanegue in [25] develops an axiomatic model connected with program verification.

The purpose of this position paper is to explore the intuitions regarding exploitations in breadth rather than in depth. We leave the development of the formalisms to the two papers mentioned above; here we discuss what could be formalized.

3 Exploitation and code reuse: a program verification perspective

Code reuse in exploitation could be seen as a trivial convenience (less attack payload to write and debug) or as a specialized trick to bypass certain protective measures that prevent execution of foreign code (e.g., non-executable stacks and heaps, or load-time code-signing checks). When seen that way, code reuse may seem a marginal phenomenon that will be eventually mitigated with further protective measures such as better ASLR, code diversity, etc. Defenders might also be tempted to apply these protections based on how desirable they think the code is for the attackers to reuse (e.g., function epilogues or indirect jumps for ROP), rather than throughout the code base.

This view would be grossly misleading. If we take the perspective that exploitation is unexpected computation, then almost *all* code that runs under conditions it was not meant to encounter—such as consuming data not conforming to the type intended for its consumption—will likely perform unexpected computation. That is, *any* code that can have its assumptions violated is potentially reusable by an exploit.

Intuitively, we might expect well-written programs to be "stable" with respect to violations of their intermediary conditions: we might expect small deviations from their component preconditions to produce only small variations in behavior.

This expectation, however, is unfounded. The fact is that correctness proofs of program verification—the best weapon we have against unexpected execution—are brittle with respect to any violations of their preconditions.

3.1 Program verification

Proving the absence of unexpected computation is, of course, the holy grail of security—so it makes sense to look at exploitation from the verification point of view. The foundations for proving correctness of computer programs were laid by the classic 1969 C.A.R. Hoare paper [14].

Hoare's formalism in a nutshell. In Hoare's construction, code Q comes with pre-conditions P, and, given that these pre-conditions hold true, after execution produces the post-conditions R (if Q terminates). Such statements $P\,\{Q\}\,R$ are written for all the elementary operations of a programming language, and combined using intuitive Boolean logic axioms, such as: if $P\,\{Q_1\}\,R_1$ and $R_1\,\{Q_2\}\,R$ hold, then $P\,\{Q_1\,;\,Q_2\}\,R$ hold. Then, if the entire program could be built up within the braces using such compositions, statement by statement, so that the condition P is empty—meaning "anything", "any circumstances"—then the program is considered proven to produce the desired post-condition, i.e., "proven correct". This construction from the elementary operations upwards is itself the proof—and corresponds directly to the more recent insights that *programs are proofs* and *propositions are types* (cf. [26]).

Code reuse vs proofs. An important property of these constructions is that they are *brittle* with respect to violations of the pre-conditions at any point in the chain. No matter if some Q is proven correct under its preconditions P, we cannot assert anything about Q's behavior under a different P', however "small" (according to some metric) the difference is.

Suppose we have a program fragment Q for which we have proven $P\,\{Q\}\,R$. Let us pose the question: what will Q compute, i.e., what set of post-conditions R it can create, if we feed it inputs that *don't* obey P?

This is exactly how an exploit (re)uses Q.

For instance, can we use Q to construct a Turing-complete computation, similar to "weird machine" constructions that reuse other code such as function epilogues, the ELF RTLD, or the x86 MMU? Can we use Q as a primitive to combine with others in an exploit? In

short, *what is the computational power of Q if we are allowed to vary its inputs arbitrarily?*

We can frame this question slightly differently. Given P, Q, $P\{Q\}R$ and some variation P' of P, what can we infer about the conditions R' such that $P'\{Q\}R'$? Do "small" variations of P by some metric result in "small" changes in R? Is there an efficient way to perform such "differential analysis" for certain classes of code Q? What can we do to make Q less amenable to yielding itself for exploit reuse?

Answering these questions may help characterize the stability of program verification proofs, and shed new light on what makes code reusable in exploits.

3.2 AEG as program verification

Automatic generation of exploits (AEG) is already being viewed as a program verification task. For example, Avgerinos et al. describe their approach as a verification task in which "the exploitability property replaces typical safety properties." [3]

The current state of exploit generation systems, however, limits them to vulnerabilities and exploits of a decade ago. Vanegue in [23] formulates the key verification problems that automated exploit generation needs to solve efficiently to compete with the modern advanced exploitation approaches.

4 Modeling exploitation

First and foremost, exploitation is a computational phenomenon. Thus it should be possible to model it the same way as we model computation, starting with the basic models such as automata, abstract machines, or rewriting systems.

Secondly, exploits are *unexpected* computations, "impossible" within the programmer's abstractions. For example, unlabeled inner statements of functions are not expected to receive control; values of variables are not supposed to be affected by writes to other unaliased variables, etc. Thus we need to model how abstractions break. Asking for formal models of how abstractions get broken may sound strange at first, but this is what we need to model exploitation.

Although some exploitation techniques involve a physical component such as affecting the state of the target with heat, magnetic fields, or radiation, a large majority of exploitation techniques use only crafted inputs. This already suggests that exploitation is somehow related to the classic models of computation theory such as automata and tape-based abstract machines, in which computation is driven by consumption of input.

An abstract automaton or machine changes its state in response to symbols of an input language, somehow delivered. For a Turing machine, these input symbols are located on the tape; for automata models, it is not specified where they exist before being fed in one by one, but we can assume a tape that moves only forward past a fixed reading-only head. Nothing other that consumption of a symbol changes the state; incoming symbols is what drives the computation.

The same is true for a typical network-bound program, except that the mechanisms delivering input characters to it are concrete (a PHY, a networking interface firmware, OS' TCP/IP stack, etc.) Notably, attackers depend on these mechanisms to operate exactly as specified, i.e., they can abstract them just as the target's developer did—unlike the targeted program's internals, where breaking programmer abstractions is the attacker's primary tool.

To model exploitation, we thus need to model exactly how crafted input payloads break abstractions.

4.1 Breaking abstractions

Classic automata definitions are quite constraining, in accordance with their purpose, to isolate the mechanical basics of computation. In a sense, there isn't much to vary about them: either the *alphabet* of symbols, or the graph of *states and transitions*, or both. Another possibility is to introduce external random mutations, e.g., random changes of symbols on the tape or random state transitions, but at least the deterministically succeeding exploits should be modeled without the use of extra randomness.

A few ways of how abstractions break reliably in exploit practice will therefore be useful to review. These ways are not just very familiar to exploiters and reverse engineers, but shape their daily practices. These examples are deliberately drawn from different technological domains and from the respectively different methods of exploitation; moreover, they are key enablers of their respective methods.

We will then discuss how these can be captured formally.

C abstraction vs machine binary execution. C statements are supposed to be indivisible units of code: control cannot transfer into the middle of a statement. C functions are similarly supposed to be indivisible unless an explicit label exists or a special construct such as `setjmp/longjmp` is used. Yet in reality any instruction of the compiled code can be the target of a jump. For example, just one or several instructions immediately before a function's epilogue can be used as an exploit gadget.

x86 instructions vs byte storage. In x86, instructions are variable-length, instruction boundaries are not marked, and instructions can start at any address regardless of alignment. The CPU will start decoding an instruction at whatever address the instruction pointer

holds. As a result, jumping inside a multi-byte instruction can yield a stream of valid instructions that were never generated as such by a compiler, nor hand-coded. Any suitable sequence of bytes can be used as a gadget, whether originally emitted as an instruction or not.

For the same reason, x86 disassembly is always speculative: unless the entry point into the code is known, multiple disassembly variants of the same bytes should be assumed.

ARM vs Thumb instruction encoding. In ARM platforms, the above instruction decoding ambiguity was made into an architectural feature: instructions are fixed-length, but come in two sizes, 32-bit (standard ARM) and 16-bit (Thumb). The least significant bit of the address to fetch an instruction from selects the decoding mode. Thus the same bytes can be interpreted as both ARM and Thumb instructions.

Constant data vs executable segments. When executing binaries produced by many C/C++ compilers (e.g., GCC) and loaded by the standard OS loader, transferring control to addresses loaded with *const* data may result in successful decoding and execution of these data bytes as instructions. The reason is that the linker groups non-writable sections of the binary such as `.rodata` with the executable `.text` into a single loadable segment, presumably to compact them in non-writable pages within the address space. This happens despite the source code clearly indicating the opposite intent.

Natural vs pivoted stacks. Compiled code of C/C++ function preambles and epilogues is assumed to deal with the program's stack, or, rather, with one of the stacks allocated for the program's threads. Such allocations are easy enough to track at either the OS or Libc, as they are made by specialized code, and their mapped memory areas are handled specially by the OS to provide automatic stack expansion. However, actual instructions cannot check these semantics, and therefore "stack" to them is wherever the stack pointer points. As a result, stacks are routinely *pivoted* by exploits from their actual locations to wherever in memory the crafted stack-imitating payload could be placed.

In-band metadata vs data. Many data structures store their data interspersed with their own management meta-data. For example, many heap implementations store their boundary tags in-band; program stacks store their control flow data in-band, and so on. This can cause confusion between data and meta-data, especially when either is corrupted. The consequences of such confusion are many and varied. Arguably, the most famous are those used by classic heap exploitation techniques that relied on the heap freelist management code acting on a corrupted Doug Lea-type boundary tag to obtain a write-four-bytes-almost-anywhere primitive [16].

Address information leaks vs OS and hardware abstractions. Software composition relies on addresses of functions and global data objects being exported by their respective software components. Since many languages require the programmer to declare the exported ones explicitly, it is natural for programmers to believe that all other addresses are *not* exposed and will not be revealed by the OS. The introduction of address randomization (ASLR) features reinforces this idea. In reality, however, OS mechanisms such as memory caching [27], memory deduplication [6], translation lookaside buffers [17], and many others prove to be a means for a program running on the same CPU or even the same machine to discover unexported addresses within another process' address space. Such information leaks came to prominence since ASLR and KASLR became widespread and countermeasures, and "almost every major memory-based exploit now relies on some form of information leak." [1]

Network packets vs PHY symbol streams. Many PHY layers, e.g., 802.11 and 802.15.4, start with a mechanism for transmitting and receiving sequences of symbols via physical processes such a modulation and demodulation of analog signals. A PHY implementation receives a stream of such symbols rather than ready frames; that is, frames must be delimited in this stream, and non-frame contents of it are discarded. When the symbols forming the *start-of-frame delimiter* (SFD) are confusable with the frame content symbols, crafted frame contents can be taken for an SFD, resulting in receipt of a crafted frame that was never sent as such (when the actual SFD is altered by noise). [12]

Streams vs their reassembly. Streams are a powerful abstraction of packet-based protocols that allows programmers to ignore specific ways that a stream's contents are segmented into packets for transmission. However, streams need to be reassembled, and reassembled versions may differ. For example, the classic [18] showed that network intrusion detection systems tended to diverge from rigorous reassembly algorithms to save time and CPU cycles, allowing attackers to manipulate their view of TCP streams as compared with the target's view and thus to hide attacks. Moreover, different OSes reassembled crafted streams with repeated TCP segments differently, some favoring the first received variant of a segment and discarding the following ones, whereas others favored the last received variant. NIDS/NIPS evasion based on differences in packet reassembly remains a basic attack technique. [20, 22]

Polyglot formats. The question of "what type is this file", or, more generally, payload, presumes that most payloads have a unique type that can be unambiguously determined, and then an appropriate handler can be called for it. Correctly determining the type can be important for security, especially when validation of an input payload according to its type performed by one component is relied upon by others. The type specification can be explicit and external, such as a MIME type specified in an HTTP response, or implicit and determined by a file or data signature, or even by a heuristic such as checking that the data contains only alphanumeric characters.[1]

However, some formats such as PDF are flexible enough to allow a single file to appear as both a valid PDF and a valid ZIP, JPEG, PNG, etc. file, as well as a variety of other formats. [2] While most of such polyglots are not a security risk, the phenomenon of disagreeing interpretations of a payload can be and has been weaponized. [15]

It should be noted that the different decodings of byte sequences into instructions are "polyglots" of the respective instruction sequences with themselves. Polyglots between *different* instruction sets, i.e., sequences that decode to correct and meaningful instructions under several architectures are also possible (as shown, e.g., in [11] for shellcode).

These are only a few examples of a how broken abstractions are used in everyday exploitation techniques. A formal view of an exploit should unify these phenomena in a non-trivial way, without losing the specifics that make them interesting to practitioners in each particular case.

4.2 Colliding abstractions

The above examples are drawn from different domains, but have one thing in common: not one but two (or more) *colliding abstractions* are involved. While one is broken, another is obeyed; the states and computation illegal under one are legal under the other.

The implicit presence of a second, obeyed abstraction is what makes exploitation a programming activity in its essence. When we speak of *exploit development* as an industrial activity, we imply that the process of creating an exploit is teachable, learnable, and has a methodology to it that can be communicated to many skilled workers; this would be impossible without suitable abstractions.

The two abstractions can be related as layers of a single stack. The "broken" abstraction then represents some semantically reasonable expectations of a developer working within a certain layer of the software stack; it just isn't borne out by the lower-layer implementation. That implementation, however, operates fully within its abstractions, nowhere breaking them. For example, even though compiled C code can be forced to jump to a mid-statement instruction, creating an execution path impossible from the C code view, this path is perfectly legal for the x86 execution model, and involves no illegal states of the processor or memory.

On the other hand, the abstractions may be competing, as they are in case of polyglots and mismatching stream reassembly: there, two same-level abstractions compete. A special case of this are de-facto optimizations of standards (including attempts to "fix" non-standard content, for which, e.g., Adobe's PDF processors are famous) lead to colliding abstractions, essentially creating two diverging definitions of what the bytes mean. Divergence may also arise from ambiguities in the standard (such as with TCP reassembly of repeating segments).

4.3 Can we formalize abstraction-breaking?

The above examples suggest that we need to start with two models: one expressing the violated abstraction, another describing the abstraction that is obeyed, such as a "leaky" implementation or a competing interpretation of the violated abstraction.

Both are driven by the same input, and the states of the broken abstraction map onto sets of states of the obeyed one—but, generally, not in reverse. We can think of the actual abstraction-breaking computation as happening simultaneously in both models up to a certain point, the same consumed symbol causing legal transitions in both, until it diverges, reaching in the contained (obeyed) abstraction a state that is no longer mappable back to the programmer-intended (violated) one. From that point on, the projection of the execution path onto the state graph of the programmer-intended model is no longer adequate for describing the state of the system.

However, when one abstraction serves as a substrate in which another is implemented—e.g., as the assembly execution model is to C's—there may exist a second mapping between these abstractions, which helps complete the definition. Namely, the mapping M from the states of the programmer-intended model to sets of states in the second one induces a partitioning on the image of M. If this partitioning can be extended to the set of *all* states beyond the image of M, then the extended partition's sets *outside* the image of M can be said to represent additional, "weird" states of the violated abstraction, through which the computation path goes.

Consider, for example, a model of $n < 256$ states, implemented as n values of a byte-sized variable in the memory of a process. The values of all other bytes in memory are irrelevant. Let's say the memory consists of N bytes, and thus has 2^{8N} states. Let the image of M consist of n subsets of $2^{8(N-1)}$ states each, and let there also be $256 - n$ subsets of $2^{8(N-1)}$ states in the induced

[1] Cf. "Rosetta Flash" `https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/` as a recent example of how such heuristics can be used by an exploit; older examples include [19].

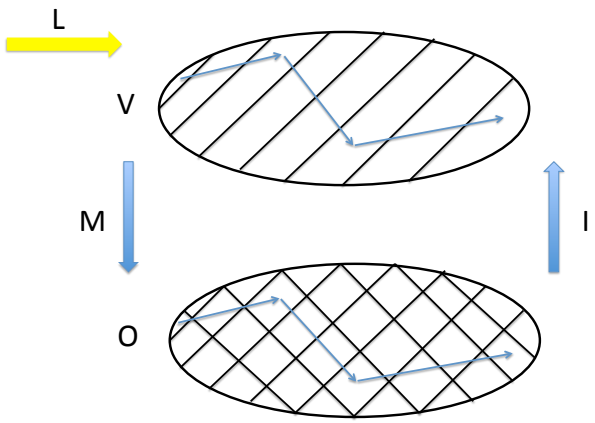

Figure 1: Programmer-intended (V) and actually obeyed (O) states and transitions.

partitioning that are not in M's image. Should the variable acquire a value outside of the n legitimate ones (say, due to an off-by-one arithmetic error), the underlying partition set could be thought of a "weird" state of the programmer-intended model, through which the actual computation goes.

Formally, we can represent it as follows. Assume there are two automata, V and O, with the state sets S_V and S_O and transition sets T_V and T_O respectively. Let them input the same language L.

The mapping M connects V and O as

$$\begin{aligned} \forall s_V \in S_V \quad & M(s_V) \subset S_O \\ \forall s_V, s'_V \in S_V \quad & M(s_V) \cap M(s'_V) = \emptyset \\ \forall t_V \in T_V \, \text{let} \, & t_O = M(t_V) \in T_O, \, t_V(s_V) = s'_V, \\ & t_O(M(s'_V)) = M(t_V(s_V)) \end{aligned}$$

A rough depiction of these can be found in Figure 1.

Assume the partitioning P on S_O that agrees with that induced by M on the image $M(S_V) \subset S_O$, i.e., $P = \{P_i\}$ such that $P_i = M(s_i)$ for some $s_i \in S_V$, $P_i \cap P_j = \emptyset$ for $i \neq j$. If this partitioning can be naturally extended to $S_O \setminus M(S_V)$, for example, if the sets P_i are natural translations of each other under some permutation of S_O (as in the above example), we'll have a partition $\bar{P} = \{\bar{P}_i\}$ of S_O, such that $P_i = \bar{P}_i$ for all $\bar{P}_i \subset M(S_V)$, and $\bar{P}_i \cap \bar{P}_j = \emptyset$ for $i \neq j$.

Then we can formally extend S_V and M with the "preimages" of $\bar{P}_k : \bar{P}_k \subset S_O \setminus M(S_V)$, writing $\bar{M}$ and $\bar{S_V}$, so that $\bar{S_V} = S_V \cup W$ where $W = \{s_W : \bar{M}(s_W) = \bar{P}_k, \bar{P}_k \subset S_O \setminus M(S_V)\}$. Similarly, we extend T_V with transitions to and from W, $\bar{T_V} = T_V \cup \{s \to w : s \in S_V, w \in W\} \cup \{w \to s : s \in S_V, w \in W\} \cup \{w \to w' : w, w' \in W\}$, as induced by the execution paths in S_O, T_O.

We can think of the preimage mapping $I : \{\bar{P}_i\} \to W$ or the more general $\bar{I} : S_O -> \bar{S_V}$, which composes with $\bar{M}$ as $\forall s \in S_V \cup W \quad \bar{I}(M(s)) = s$. The mapping $\bar{I}$ is, in these terms, the *essence* of V's implementation by O (hence the choice of "I").

With these extensions, we can describe the computations taken in machines V and O on an input $l \in L$ as the sequences of states and transitions $(s_{O1}, t_{O1}, \ldots, s_{Oi}, t_{Oi}, \ldots)$ for O and $(s_1, t_1, \ldots, s_i, t_i, \ldots)$ for V, where $s_i \in S_V, t_i \in T_V$ for all $i < i_0$, $s_{i_0} \in W, t_{i_0} \in \bar{T_V}$, and $s_i \in S_V \cup W, t_i \in \bar{T_V}$ for $i > i_0$. For these two sequences, $M(s_i) = s_{Oi}$, $M(t_i) = t_{Oi}$ for all i.

Whether this construction appears natural or not, depends on the internal structure of the underlying state space S_O and its partitioning P induced by M. Any natural symmetries on the subsets of S_O that map $\{P_i\}$ around and allow to extend it to a partition $\bar{P}$ of the entire S_O provide this structure. Then, on any path in the S_O state space we can lift any state $s \in S_O \setminus M(S_V)$ to the preimage w_{P_k} such that $\bar{M}(w_{P_k}) = P_k$.

The extension of the transition set T_V is tied with those of S_V and M. Specifically, for a transition $s_O \to s'_O, s_O \in M(S_V), s'_O \in S_O \setminus M(S_V), M(s_V) = s_O, \bar{M}(w) = s'_O$ we put $\bar{t} \in \bar{T_V}$ defined as $\bar{t} : s_V \to w$.

This construct describes an approach to extending the top-level vulnerable abstraction with additional "weird" states to describe the exploit combination that leaves the abstraction's original state space—and does not need the full complexity of the underlying implementation abstraction to describe itself. This is important, because exploitation, as any programming activity, tends to operate with the most economical descriptions of the programming model.

We further see that the key role in this construct is played by the structure of the obeyed abstraction's state space. Without such structure, extending the broken abstraction's states to represent the computation's path would be contrived and thus not useful. This structure, however, comes from the semantics of the underlying abstraction—which our general notation does not capture, but specific models such as [10, 25] will.

5 Conclusion

Modern exploits are complex programs that rely on breaking abstractions to reuse implicitly existing or emergent features of the target architecture. These emergent architectures on which the exploits run should be studied in their own right if we hope to achieve preventing unexpected computation as a phenomenon rather than merely mitigating known exploit techniques in an endless game of whack-a-mole. To do so, we need to develop abstractions that describe exploitation from the computational point of view—just as in computation theory we developed simple models of computation to answer the questions of what computers can and cannot do. In security, we now face the same question for every bit of code that may receive attacker-crafted input.

Acknowledgments

The authors gratefully acknowledge many helpful discussions with Felix 'FX' Lindner, Julien Vanegue, and Thomas Dullien about the phenomenon of exploitation and its generalizations. The authors are also greatly indebted to Meredith L. Patterson and Len Sassaman who pioneered the language-theoretic and computation-theoretic view of security and exploitation.

Literature

[1] Aaron Adams. Exploitation Advancements. Research Insights, vol. 7, NCC Group, 2015.

[2] Ange Albertini. Abusing file formats; or, Corkami, the novella. *PoC||GTFO*, 7, March 2015.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2):74–84, 2014.

[4] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop on Offensive Technologies*. USENIX, 2013.

[5] Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, May 2014.

[6] Erik Bosman, Kaveh Razavi, Herbert Bos, , and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *37th IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2016.

[7] Sergey Bratus. What Hacker Research Taught Me. Recurity Labs Security Seminar, Berlin, December 2010.

[8] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login:*, December 2011.

[9] Thomas Dullien. Exploitation and State Machines: Programming the "Weird Machine", revisited. `http://www.immunityinc.com/infiltrate/presentations/Fundamentals_of_exploitation_revisited.pdf`, April 2011. Infiltrate Conference.

[10] Thomas Dullien. Fundamentals of Exploitation. in preparation, 2016.

[11] eugene. Architecture Spanning Shellcode.

[12] Travis Goodspeed, Sergey Bratus, Ricky Melgares, Rebecca Shapiro, and Ryan Speers. Packets in Packets: Orson Welles' In-Band Signaling Attacks for Modern Radios. In David Brumley and Michal Zalewski, editors, *5th USENIX Workshop on Offensive Technologies*, pages 54–61. USENIX, August 2011.

[13] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks: Stealing More Pie Without Touching the Sill. *J. Comput. Secur.*, 22(4):567–599, July 2014.

[14] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[15] Jay Freeman (saurik). Exploit (& Fix) Android "Master Key"; Android Bug Superior to Master Key; Yet Another Android Master Key Bug. `http://www.saurik.com/id/17`; `http://www.saurik.com/id/18`; `http://www.saurik.com/id/19`, August 2013.

[16] jp. Advanced Doug Lea's malloc Exploits. *Phrack* 61:6. `http://phrack.org/issues.html?issue=61&id=6`.

[17] Sangho Lee, Taesoo Kim, and Yeongjin Jang. Breaking KASLR with Intel TSX. In *Black Hat USA*, August 2016.

[18] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998. `http://insecure.org/stf/secnet_ids/secnet_ids.html`.

[19] rix. Writing ia32 alphanumeric shellcodes. Phrack 57:15, November 2001. `http://phrack.org/issues/57/15.html`.

[20] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion without Altering Traffic. In *IEEE Symposium on Security and Privacy*, pages 44–61, 2003.

[21] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *7th USENIX Workshop on Offensive Technologies*. USENIX, 2013.

[22] Sumit Siddharth. Evading NIDS, revisited. `http://www.symantec.com/connect/articles/evading-nids-revisited`, December 2005. (updated Nov 2010).

[23] Julien Vanegue. The Automated Exploitation Grand Challenge. H2HC conference, Sao Paulo, Brazil, October 2013. `http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf`.

[24] Julien Vanegue. The Weird Machines in Proof-Carrying Code. In *1st IEEE Language-theoretic Security & Privacy Workshop*, 2014.

[25] Julien Vanegue. Provably Unsafe Programs. in preparation, 2016.

[26] Philip Wadler. Propositions as Types. *Communications of the ACM*, 58(12):75–84, 2015.

[27] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732. USENIX Association, August 2014.

Dr. Sergey Bratus Sergey Bratus is a Research Associate Professor of Computer Science at Dartmouth College. Sergey is a member of the LangSec.org project that seeks to eliminate large classes of bugs related to input handling, and worked with industrial control systems stakeholders to develop architectural protections for ICS/SCADA systems and protocols. He has a Ph.D. in Mathematics from Northeastern University.

Address: Dartmouth College, Institute for Security, Technology, and Society, Hanover, NH 03755, E-Mail: sergey@cs.dartmouth.edu

Dr. Anna Shubina Anna Shubina is a post-doctoral research associate at the Dartmouth Institute for Security, Technology, and Society. Anna was the operator of Dartmouth's Tor exit node when the Tor network had about 30 nodes total. She currently manages the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research.

Address: Dartmouth College, Institute for Security, Technology, and Society, Hanover, NH 03755, E-Mail: ashubina@cs.dartmouth.edu