DOI:10.1145/2560217.2560219

The idea is to identify security-critical software bugs so they can be fixed first.

BY THANASSIS AVGERINOS, SANG KIL CHA, ALEXANDRE REBERT, EDWARD J. SCHWARTZ, MAVERICK WOO, AND DAVID BRUMLEY

Automatic Exploit Generation

ATTACKERS COMMONLY EXPLOIT buggy programs to break into computers. Security-critical bugs pave the way for attackers to install trojans, propagate worms, and use victim computers to send spam and launch denial-of-service attacks. A direct way, therefore, to make computers more secure is to find security-critical bugs before they are exploited by attackers.

Unfortunately, bugs are plentiful. For example, the Ubuntu Linux bug-management database listed more than 103,000 open bugs as of January 2013. Specific widely used programs (such as the Firefox Web browser and the Linux 3.x kernel) list 7,597 and 1,293 open bugs in their public bug trackers, respectively.[a] Other projects, including those that are closed-source, likely involve similar statistics. These are just the bugs we know; there is always the persistent threat of zero-day exploits, or attacks against previously unknown bugs.

Among the thousands of known bugs, which should software developers fix first? Which are exploitable? How would you go about finding the unknown exploitable ones that still lurk?

Given a program, the automatic exploit generation (AEG) research challenge is to both automatically find bugs and generate working exploits. The generated exploits unambiguously demonstrate a bug is security-critical. Successful AEG solutions provide concrete, actionable information to help developers decide which bugs to fix first.

Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar "History of AEG"). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an "exploitability" property, and the "verification" process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.

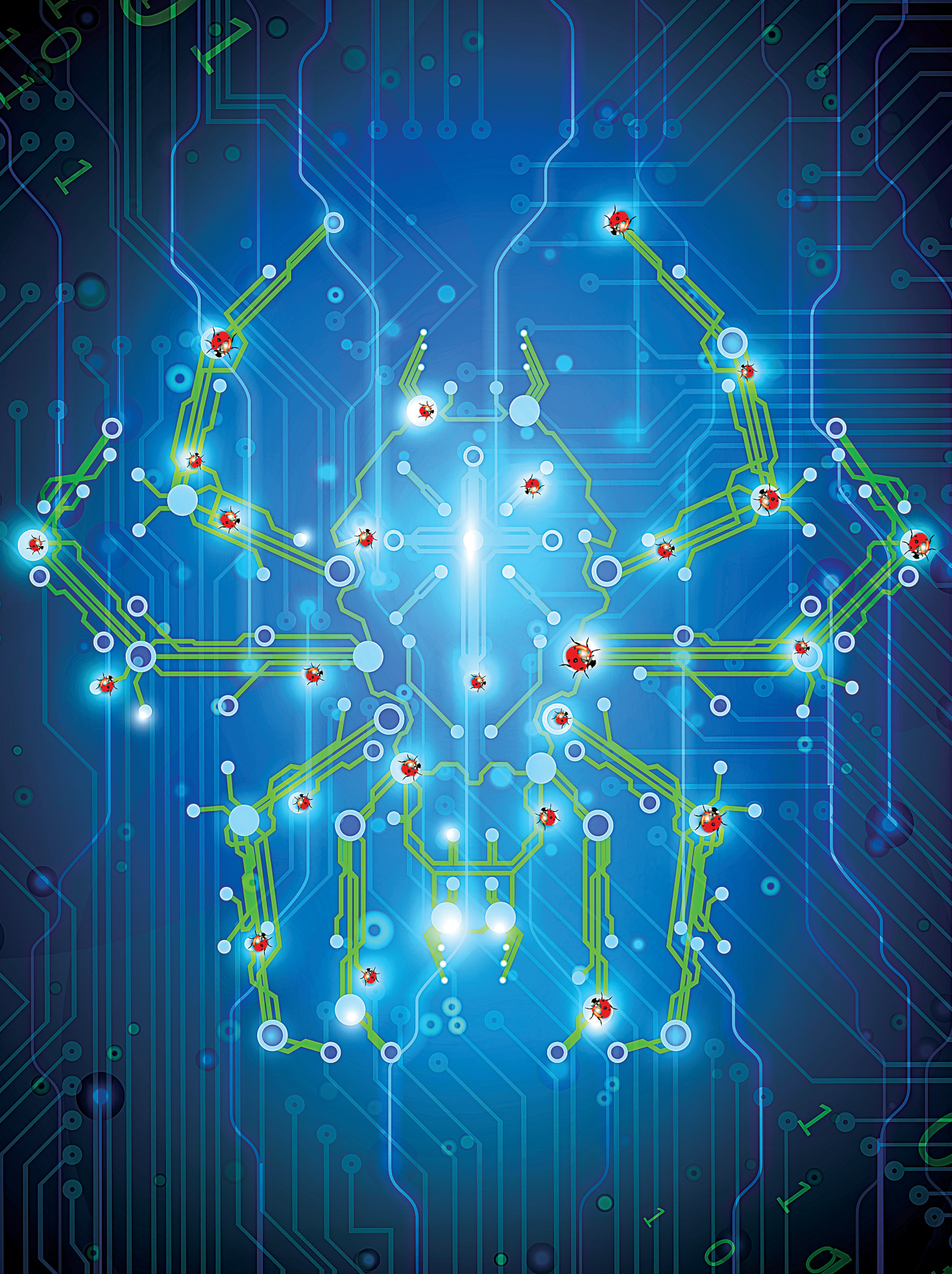
Verification involves many well-known scalability challenges, several of which are exacerbated in AEG. Each new branch potentially doubles the number of possible program paths, possibly leading to an explosion of paths to check for exploitability. Traditional verification takes advantage of source code, models, and other abstractions to help tackle the state explosion and scale. Unfortunately, ab-

» key insights

- **This research formalizes the notion of an exploit, allowing for automated reasoning about exploitation.**
- **The technology can be used to identify and prioritize security-critical bugs.**
- **Improvements for verifying programs safe may also lead to improvements for automatically generating exploits.**

a All bug counts exclude bugs tagged as "wishlist," "unknown," "undecided," or "trivial."

IMAGE COLLAGE BY IWONA USAKIEWICZ/ANDRIJ BORYS ASSOCIATES



stractions often leak by not perfectly encapsulating all security-relevant details, and the leaky points tend to affect the quality of security analysis. For example, writing 12B to an array declared to be 11B long is wrong in C but is also unlikely to be exploitable because most compilers would pad the array with extra bytes to word-align memory operations.

In order to provide high fidelity, most AEG work analyzes raw executable code. Executable code analysis is needed because many exploits rely on low-level details that are abstract in source code (such as CPU semantics and memory layout). Executable code analysis is also attractive because it is widely applicable; users typically have access to the executable code of the programs they run (as opposed to source code) and thus can audit the code for security-critical bugs.

Throughout this article, we focus on AEG as a defensive tool for prioritizing exploitable bugs. However, we are also cognizant of the obvious offensive computing implications and applications as well. Governments worldwide are developing computer-warfare capabilities, and exploits have become a new type of ammunition. At present, exploit generation in practice is mostly a manual process. Therefore, techniques that help reduce the time and effort for exploit generation can potentially affect a nation's operational capabilities. AEG research is in its infancy and not yet at the point of automatically churning out weapons-grade exploits for an arbitrary program. Most reported research results generate exploits against bugs up to a few thousand lines deep in execution and for relatively straightforward bugs, while typical offensive needs include exploits for complicated bugs and large programs like Internet Explorer and Adobe Reader. Nonetheless, current AEG results show promise, and a conservative defensive security position must consider the possibility of real-world offensive AEG capabilities.

This article describes our AEG research at Carnegie Mellon University, its successes, as well as its current limitations. We focus primarily on control-flow hijack exploits that give an attacker the ability to run arbitrary code. Control-flow hijacks are a serious threat to defenders and coveted by attackers.[3,35] Although most current research focuses on control-flow hijacks due to their immediate danger, AEG is not limited to only this class of attacks. Exploitable bugs are found in programs in all languages, and the verification-based approach to AEG still applies.

Exploiting Programs

Suppose a developer is interested in finding and fixing exploitable bugs in the `/usr/bin` directory of the latest Debian operating system. For instance, in June 2012 we downloaded the then-current Debian 6.0.5, with (in our installation) 1,168 executables in `/usr/bin` to analyze for exploitable bugs.

A typical approach to finding exploitable bugs is to first find them and then determine which ones are exploitable. One popular way to find bugs is to perform "black-box fuzzing." Fuzzing is a program-testing technique that runs a program on inputs from a fixed alphabet, often either modifying at random a known input or trying extreme values (such as 0 and the native maximum integer), and the "black-box" refers to the program itself, which is not analyzed at all. The fuzzer chooses the inputs and observes the program, looking for hangs, crashes, buggy outputs, or other indications of a bug.

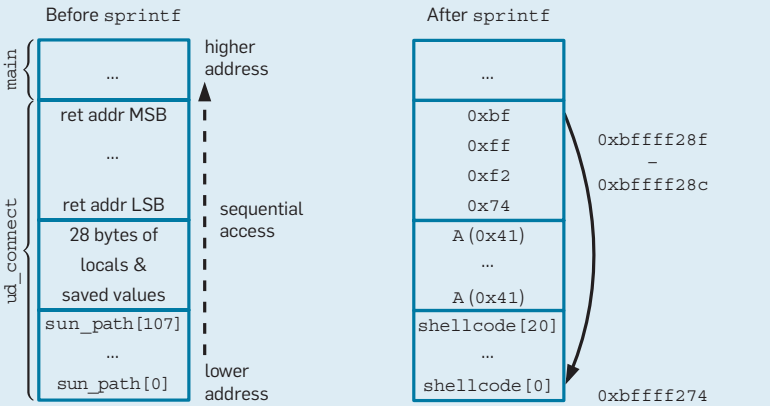
We fuzzed each program using the following script:

```
for letter in {a..z} {A..Z}; do
 timeout -s 9 1s <program>
 -$letter <path>
done
```

The script tries all single-letter com-

Our running example of a buffer overflow in `acpi-listen`.

```
 1. int main(int argc, char **argv) {
 2.   char *name; int i;
 3.   for (;;) {
 4.     i = getopt(argc, argv, "c:s:t:vh");
 5.     if (i == -1) break;
 6.     switch (i) {
 7.       case 'c': ...; break;
 8.       case 's': name = optarg; break;
 9.       ...
10.     }
11.   }
12.   sock_fd = ud_connect(name);
13.   ...
14. }
15. int ud_connect(const char *name) {
16.   int fd;
17.   struct sockaddr_un {
18.     sa_family_t sun_family;
19.     char sun_path[108];
20.   } addr;
21.   ...
22.   sprintf(addr.sun_path, "%s", name);
23.   ...
24.   return fd;
25. }
```




```
00000000  31 c9 f7 e1 51 68 2f 2f  73 68 68 2f 62 69 6e 89  |1...Qh//shh/bin.|
00000010  e3 b0 0b cd 80 41 41 41  41 41 41 41 41 41 41 41  |.....AAAAAAAAAA|
00000020  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
*
00000080  41 41 41 41 41 41 41 41  74 f2 ff bf              |AAAAAAAAt...|
```

mand-line options from a to Z, followed by a valid 6,676B filename. The `timeout` command limited total execution time to one second, after which the program was killed.

The script took about 13 minutes to fuzz all programs on our test machine, yielding 756 total crashes. We identified 52 distinct bugs in 29 programs by analyzing the calling context and faulting instruction of each crash. Which bugs should a developer fix first? The answer is the exploitable ones. For now, we forgo several important issues relevant in practice we tackle later (such as whether the buggy program is a realistic attack target and whether additional operating system defenses would protect the otherwise exploitable program from attack).

We first describe simple manual exploit generation to introduce terminology and give a flavor of how exploits work. We focus on control-flow hijack exploits, which have been a staple class of exploits in the computer-security industry for decades.[3,35] Well-known examples of control-flow hijacks range from exploits in the Morris worm in 1988 to the more recent Stuxnet and Flame worms (though the latter exploits are much more complicated than those described here).

The figure here shows a bug discovered in `acpi_listen` (now patched in Debian testing) we use as our running example. A buffer overflow occurs on line 22. The program reads in a command-line argument; if it is `-s` (line 8), it assigns the subsequent argument string to the name variable. On line 22, the `sprintf` function copies name into `sun_path`, a field in a local instance of the networking `sockaddr_un` data type, a standard data structure in Unix for sockets.

The bug is that `sun_path` is a fixed-size buffer of 108B, while the command-line argument copied through name into `sun_path` can be any length. The C standard says the execution behavior is undefined if more than 108B are written. When executed, something will happen; with the fuzzing script described earlier, the program crashed. Unfortunately, this crashing bug can be exploited.

All control-flow hijack exploits have two goals: hijack control of the instruction pointer (IP) and then run an attacker's computation. For `acpi_listen`, some of the details an attacker must understand in-depth include: the hardware execution model (such as how instructions are fetched from memory and executed; how function calls are implemented; how writing outside the allocated space can hijack control of the IP; and how to redirect the IP to run the attacker's code). Since any discussion of creating exploits against vulnerable C programs assumes a basic understanding of these facts, we offer the following overview.

The twist is we replace typical safety properties with an "exploitability" property, and the "verification" process becomes one of finding a program path where the exploitability property holds.

During runtime, computer hardware implements a fetch-decode-execute loop to run a program. The hardware maintains an IP register that contains the memory address of the next instruction to be executed. During the fetch phase, the hardware loads the data pointed to by the IP register. The data is then decoded as an instruction that is subsequently executed. The IP is then set to the next instruction to be executed. Control is hijacked by taking control of the IP, which is then used to fetch, decode, and execute the attacker's computation.

A straightforward exploit for `acpi_listen` hijacks control by overwriting data used to implement function returns. Exploits can also overwrite other control data (such as function pointers and the global offset table, as in Muller[29]), but we omit these details here. Function calls, returns, and local variables are not supported directly by hardware. The compiler implements the semantics of these abstractions using low-level assembly instructions and memory. An attacker must be proficient in many details of code execution (such as how arguments are passed and registers are shared between caller and callee). For simplicity, we assume a standard C calling convention known as `cdecl`. Functions using it implement a stack abstraction in memory where functions push space for local variables, arguments to future calls, and other data onto the stack immediately after being called. A function return pops the allocated space off the stack. Thus, the stack grows a bit for each call and shrinks a bit on each return.

When *f* calls *g*, *f* first puts *g*'s arguments onto the stack, then invokes *g*, typically through a `call` assembly instruction. The semantics of `call` includes pushing *f*'s return address onto the stack; that is, the address in *f* where execution (normally) continues once *g* terminates. Upon entrance, *g* creates space for its variables and other run-time information (such as saved register values). After *g* completes, *g* returns control to *f* by shrinking the created stack space for *g* and popping off the saved address into the IP register, typically through a `ret` instruction. A critical detail is that the popped value, regardless of whether it was the original value pushed by *f* or not, is used as the address of the next instruction to execute. If an attacker can overwrite that address, the attacker can gain control of execution.

The stack frame just before `sprintf` is called on line 22 in the Figure. The flow of execution for creating the depicted stack includes six steps:

Return address pushed onto the stack. When `main` called `ud_connect`, `main` pushed the address of the next instruction to be executed (corresponding to line 13) onto the stack;

Control transfer. `main` transferred control to `ud_connect`;

Local variable space allocated. `ud_connect` allocated space for its local variables. On our computer, 108B were allocated for `sun_path` and an additional 28B for other data (such as additional local variables and saved register values);

Function body executed. The body of `ud_connect` ran. When `sprintf` is called, a similar flow pushes a new return address on the stack and new space onto the stack for `sprintf`'s local variables;

Local variable space deallocated. When `ud_connect` returns, it first deallocates the local variable space, then pops off the saved return address into the IP register; and

Return to caller. Under normal operation, the return address points to the instruction for line 13, and `main` resumes execution.

The crux of a control-flow hijack is that memory is used to store both control data (such as return addresses) and program-variable values, but the control data is not protected from being overwritten in a variable update. Control-flow hijacks are an instance of a channeling vulnerability that arise when the control and data planes are not rigorously separated. For this particular example, an out-of-bound write can clobber the return address. When `sprintf` executes, it copies data sequentially from `name` up the stack, starting from the address for `sun_path`, as shown. The copy stops only when a zero integer, or ASCII NULL, is found, which is not necessarily when `sun_path` runs out of space. A long name will clobber the saved local variables and eventually the saved return address. Since an attacker controls the values in `name`, and those values overwrite the return address, the attacker ultimately controls which instructions are executed when `ud_connect` returns.

Governments worldwide are developing computer-warfare capabilities, and exploits have become a new type of ammunition.

Attackers must analyze the program to figure out exactly how many bytes to write, what constraints may be on the bytes, and what would be a good value with which to overwrite the return address. For `acpi_listen`, a string of length 140 will overwrite the return address. The first 108B will be copied into space allocated for `sun_path`. The next 28B on the stack are intended to hold local variables and saved register values. The final 4B overwrite the saved return address.

When `ud_connect` returns, the overwritten return address is popped off the stack into the IP register. The machine continues executing the instruction starting at the overwritten address. While this example overwrites the return address, a variety of other control data structures can be used to seize control; examples include function pointers, heap metadata, and C++ virtual function tables.

Control is typically hijacked to run an attacker-supplied computation. The most basic attack is to inject executable code into the vulnerable process. More advanced techniques (such as command injection, return-to-libc, and return-oriented programming) are also possible[29,33] (and in some cases can be automated as well[31]), but we omit such discussion here.

A natural choice for the computation is to execute the command-line shell `/bin/sh` so the attacker is able to subsequently run any command

with the privileges of the exploited process. In fact, executing a shell is so popular that colloquially any attacker code is called "shellcode." A classic approach is to give executable code as input to the program and redirect control flow to the given executable code. The executable code itself can be created by mimicking the assembly for `execve("/bin/sh", args, NULL)`. Attackers introduce the shellcode to the vulnerable program as a normal string program input that is eventually decoded and executed as code.

The final step of the attack is to overwrite the return address with the address of the shellcode. On our machine, `sun_path` is at memory address `0xbffff274`. The complete exploit for `acpi_listen` (generated automatically by our AEG tools) is shown in the figure, where:

Shellcode. The first bytes of the command line argument are the shellcode; the shellcode is 21B, and, in this case, the first 21B are copied into bytes 0–20 of `sun_path`;

Padding. The next 115B of input can be any non-zero, or non-NULL ASCII, value; the bytes are copied into bytes 21–107 of `sun_path` and the additional space for other locals; and

Shellcode address. The last 4B of input are the hex string `0x74 0xf2 0xff 0xbf`. They overwrite the return address. When the return address is popped, the bytes become the address `0xbffff274` (because x86 is little endian), which is the address of the shellcode after it is copied to `sun_path`.

The figure shows the stack frame after supplying this string as a command-line argument following `-s`. When `ud_connect` returns, the address `0xbffff274` is popped into the IP register, and the hardware fetches, decodes, and executes the bytes in `sun_path` that, when interpreted as executable code, runs `/bin/sh`. When the shellcode runs, the attacker is able to run any command with the same privileges as the exploited program.

Research Vision

Manual exploit generation requires a developer to reason about an enormous number of details (such as size of the stack, location of control flow critical data, like return address, and precise semantics of each instruction). Our research vision is to automate it.

AEG uses verification techniques to transform the process of finding and deciding exploitability to reasoning in logic. At a high level, AEG consists of three steps: It first encodes what it means to exploit a program as a logical property; it then checks whether the exploitability property holds on a program path; and finally, for each path the property holds, it produces a satisfying input that exploits the program along the path.

These steps are the cornerstones of AEG research. First, what exploitability properties do we encode, and how? In industry, an exploit could mean control-flow hijack, while an intelligence agency might also include information disclosures, and a safety board could include denial of service for critical services. Any single property may have many encodings, with some more efficient for automated tools to check than others. Second, what techniques and algorithms should a programmer employ to check a program? The general problem of checking programs for properties is called "software model checking,"[24] encompassing a number of techniques (such as bounded model checking, symbolic execution, and abstract interpretation). Third, what does it take to implement real systems, and how do these systems perform on real software?

The theory of AEG can be described with a small number of operations on a well-defined programming language that interacts with its environment in a few predicable, easy-to-model ways. However, a real system must also contend with hundreds of CPU instructions and the tricky and complex ways programs interact with their environments. Sometimes even pedestrian yet necessary details are difficult to get right; for example, it took our team almost a year to stop finding bugs in our internal semantics for the x86 shift instructions (such as `shl`). The developers of Microsoft's SAGE tool reported similar difficulties for the same instructions.[4]

Current AEG research primarily uses symbolic execution[25] to check program paths for exploitability properties. At a high level, symbolic execution represents all possible inputs as a set of symbolic input variables. Symbolic execution then picks a program path through a predefined path-selection algorithm. The path is then "executed," except, instead of executing on a real, concrete input, a symbolic input stands in for any possible concrete value. Symbolic execution builds up a path formula in terms of the symbolic inputs based on the instructions executed. The path formula is satisfied, meaning made true, by any concrete input that executes the desired path. If the path formula is unsatisfiable, there is no input that executes the path, and the path is called infeasible. The satisfiability check itself is done through automated solvers (such as Satisfiability Modulo Theories, or SMT).[15] By construction, free variables correspond to program inputs, and any satisfying assignment of values to free variables (called a model) is an input that executes the selected path. SMT solvers enumerate satisfying answers when needed.

In `acpi_listen`, the symbolic inputs are the first two arguments `argv[1]` and `argv[2]`. (Although we have shown source code for `acpi_listen` for clarity, our AEG tool Mayhem requires only the program executable.[12]) Executing the `-s` option program path generates the constraint that the first 3B of `argv[1]` correspond to the NULL-terminated string `-s`. At each subsequent branch point, symbolic execution adds more constraints to the formula. Next, `acpi_listen` calls `sprintf`, which copies bytes from name to `addr.sun_path` until it encounters a NULL character. Symbolic execution captures this logic by adding the constraint that each copied byte is non-NULL. Symbolically executing the `-s` program path where `argv[1]` is three symbolic bytes and `argv[2]` is 140 non-`NULL` symbolic bytes generates the constraints:

$$\texttt{argv[1][0:2]= "-s"} \wedge \forall i \in [0,139]. \texttt{argv[2][i]} \neq 0 \wedge \texttt{argv[2][140]}=0 \quad (1)$$

Note that a formula may have many satisfying answers; for example, bytes 0–139 of `argv[2]` can be "A," "B," or any other non-NULL character.

Each feasible path can be checked for exploitability by adding a set of con-

History of AEG

Symbolic execution was invented around 1975 independently by several researchers.[5,23,25] Around 2005, the field exploded. Hundreds of papers have now been published describing advanced techniques and applications; see Cadar and Sen[11] for a description and the main challenges of symbolic execution. Modern tools (such as KLEE,[9] EXE,[10] SAGE,[20] and others[7,13,32,37]) find inputs that can crash or hang a system. Such inputs may well be viewed as exploits in safety-critical systems where uptime is critical. More generally, work in symbolic execution is directly applicable to making AEG more efficient. As of 2012, most symbolic-execution work followed one execution path at a time. Since then, more work has looked at generalizing over multiple paths (such as to loops[30]). Others have also investigated alternatives to symbolic execution that tame path explosion (such as Brumley and Jager[6] and Flanagan and Saxe[16,18,26]). More generally, any verification technique that can produce example inputs (such as bounded model checking) is likely usable for AEG.

Modern AEG research dates to at least Ganapathy et al.,[17] who explicitly connected verification to exploit generation, modeling how format string specifiers are parsed by functions like `printf` that take a variable number of arguments and use the model to automatically generate exploits. They also demonstrated automatically generating an exploit against a key integrity property for a cryptographic co-processor.[17] However, they considered only API-level exploits, which do not include running shellcode or the conditions necessary to reach a vulnerable API call site.

In 2007, Medeiros[28] and Grenier et al.[21] proposed techniques based on pattern matching for AEG.

In 2008, Brumley et al.[8] developed automatic patch-based exploit generation (APEG). The APEG challenge is, given a buggy program P and a patched version P', generate an exploit for the bug present in P but not present in P'. The idea is the difference between P and P' reflects where the original bug occurs and under what conditions it might be triggered. Attackers have long known the value of analyzing patches to find non-public bugs; for example, attackers have been known to joke Microsoft's "patch Tuesday" is followed by "exploit Wednesday." Our techniques automatically found the differences between P and P' and generated inputs that triggered the bugs in P using symbolic execution. One main security implication is that attackers can potentially use APEG to exploit bugs before patches can be distributed to a large number of users. We generated exploits for five Microsoft security patches, including triggering an infinite loop in the TCP/IP driver and stealing files on Microsoft Web servers. One limitation was that our work on APEG only proposed, but did not implement, techniques for executing shellcode for memory-safety bugs.[8]

Heelan's 2009 thesis[22] was the first to comprehensively describe and implement techniques for automatically generating control-flow hijack exploits that execute shellcode. In Heelan's problem setting, the attacker is given an input that executes an exploitable program path, and the goal is to output a working control-flow-hijack exploit. This setting is the same as in our running example where we first fuzzed to find bugs, then checked exploitability. Heelan proposed using symbolic execution and taint analysis to derive the conditions necessary to transfer control to shellcode and demonstrated a tool that produced exploits for several synthetic and for one real vulnerability. He also used a technique called return-to-register to improve exploit robustness. Heelan's thesis also presented a history of AEG work through 2009.

In 2011, we proposed AEG techniques that find bugs and generate exploits, demonstrating them on 16 vulnerabilities.[1] The initial work performed symbolic execution on source code to find bugs, then used dynamic binary analysis to generate control-flow hijack exploits. Included were a number of optimizations for searching the state space (such as preconditioned symbolic execution and the buggy-path first optimizations discussed earlier). In 2012, we introduced Mayhem, a tool and set of techniques for AEG on executable code.[12] With Mayhem, we proposed techniques for actively managing symbolically executed program paths without exhausting memory and reasoning about symbolic memory addresses efficiently. Both papers[1,12] targeted control-flow hijacks for buffer overflows and format-string vulnerabilities. Mayhem generated exploits for seven Windows and 22 Linux vulnerabilities. Disregarding one long-running outlier, the average exploit-generation time in all experiments was 165 seconds. As of July 2013, Mayhem was able to generate exploits for buffer overflows, format strings, command injection, and some information-leak vulnerabilities.

AEG[1] and Mayhem[12] were designed to demonstrate a bug is exploitable but do not try to bypass defenses that may otherwise protect a system. In 2011, we proposed techniques for bypassing the DEP and ASLR defenses implemented in Windows 7 and Linux, as well as exploit hardening and maintenance.[31]

straints that are satisfied only by exploiting inputs. Most research tackles control-flow hijack exploits, where the exploitability constraints specify the IP register holds a value that corresponds to some function f of user input i (such as, f may be a call to `tolower` on the input i) and the resulting IP points to shellcode:

$$\texttt{IP=f(i)}\wedge\texttt{mem[IP]=}\langle\texttt{shellcode}\rangle \quad (2)$$

Now let a_r be the memory address for the return address and a_s be the address of our shellcode. The full formula to reach and exploit the `acpi _ listen` bug is:

$$\texttt{(Equation 1)}\wedge\texttt{mem[ar]=as}\wedge\texttt{mem[as:as+len(shellcode)-1]=}\langle\texttt{shellcode}\rangle \quad (3)$$

The $\texttt{mem}[a_r]$ constraint requires the return address to contain the address of the shellcode a_s. The final constraint requires the shellcode to start at address a_s. The variable a_s is left unconstrained since the shellcode could potentially be placed anywhere in memory. In our experiment, our AEG tool Mayhem[12] found the exploitable path and solved the exploitability formula in 0.5 seconds. Mayhem is also able to enumerate satisfying answers to automatically generate multiple exploits.

Managing state explosion. AEG is a type of software verification, albeit for a very special property. As such, it inherits benefits but also well-known scalability challenges (such as path explosion and the NP-hardness of solving SMT queries in general). They are often

amplified in AEG because AEG techniques reason about both low-level code and large inputs, along with a few abstractions. However, specific characteristics of AEG also afford researchers unique opportunities.

Consider the affect of path prioritization on this program:

```
int x = get_int();
if((x % 2) == 0) {
  if(x > 10) vuln1();
  else if(x == 3) vuln2();
  else safe();
} else { safe(); }
```

Let x_o be ao explore the program and find the vulnerability:

$$(x_0 \% 2) = 0 \wedge \neg(x_0 > 10) \wedge \neg(x_0 = 3)$$
$$(x_0 \% 2) = 0 \wedge \neg(x_0 > 10) \wedge x_0 = 3$$
$$\neg((x_0 \% 2) = 0)$$
$$(x_0 \% 2) = 0 \wedge x_0 > 10$$

The first formula for the first path is satisfiable (such as when $x_o = 4$), indicating the path can be executed but is safe (unexploitable). The second formula corresponds to the infeasible path up to `vuln2()` and is unsatisfiable because the constraint (x_o % 2) = 0 and x_o =3 cannot both be true simultaneously. Since `vuln2` will never be executed, it can never be exploited. The third formula corresponds to a feasible, safe path. Only the fourth formula corresponding to the path up to `vuln1()` is satisfiable, where a satisfying assignment (such as x_o =42) corresponds to an exploit. In general, the number of paths and formulas is infinite for programs with loops and exponential in terms of number of branches for any acyclic portion, making effective path selection a fundamental issue in AEG research.

Path-selection heuristics guide execution so vulnerable paths are selected early in exploration. Symbolic execution research is filled with a variety of approaches. For example, KLEE has options for depth-first traversal of the control-flow graph, as well as a randomized strategy.[9] Microsoft uses generational search,[20] which prioritizes symbolically executing program paths that branch off a known path taken by a fixed concrete seed input. Godefroid et al.'s research[20] suggests generational search is more effective than either breadth-first search or depth-first search.

A sound AEG technique says a bug is exploitable if it really is exploitable, while a complete technique reports all exploitable bugs.

Two techniques that proved effective in our experiments at Carnegie Mellon are "preconditioned symbolic execution" and "buggy-path first."[1] Preconditioned symbolic execution first performs lightweight analysis to determine the necessary conditions to exploit any lurking bugs, then prunes the search space of paths that do not meet these conditions. For example, a lightweight program analysis may determine the minimum length input string needed to trigger possible buffer overflows, and paths corresponding to inputs smaller than the minimum length can be pruned or skipped.

The idea of buggy-path first is that any bug is a sign of programmer confusion, increasing the likelihood of an exploitable bug being nearby. For example:

```
char buf[1024];
memset(buf, 0, strlen(input));
...
strncpy(buf, input,
strlen(input));
```

The second line contains a mistake where potentially more than 1,024B of `buf` are zeroed. This bug would likely not lead to a control-flow hijack, but does signal confusion that the length of input is somehow related to the size of `buf`. Buggy-path first would prioritize further exploration of the buggy path over other possible paths and thus discover the subsequent exploitable code more quickly in our tests. Note that a unique aspect of buggy-path first is that execution continues under the assumption the bug has been triggered (such as in the example when nearby stack variables may have been zeroed inadvertently).

A second core challenge of AEG research is optimizing SMT satisfiability checks. In theory, each satisfiability check is an NP-hard problem instance, but in practice many queries are resolved quickly. For example, in an experiment involving 5.6 million SMT queries, 99.98% of all solved queries took one second or less. Domain-specific optimizations in symbolic execution (such as arithmetic and logical simplifications, strength reduction,

concrete execution, and caching) all help speed queries.[9,10,13,19,32]

In 2006 when we started using symbolic execution and SMT solvers, we treated the SMT solver as a black box, focusing only on the symbolic executor. In hindsight, that approach was naive. In our research group we now believe it is more fruitful to view the SMT solver as a search procedure and use optimizations to guide the search. For example, one recurring challenge in AEG is checking satisfiability of formulas that operate on memory with symbolic memory addresses. A symbolic memory address occurs when an index into an array or memory is based on user input, as in:

```
...; y=mem[i%256]; if(y==2) vuln(); ...
```

Without more information, the SMT solver must do a case split over all possible values of *i* that may reach downstream statements (such as `vuln`). Case splits can quickly push an SMT solver off an exponential cliff. Symbolic memory references often crop up in commonly occurring library calls (such as conversion functions like `tolower` and `toascii`) and parsing functions (such as `sscanf`). Many symbolic executors mitigate the case split by concretizing symbolic addresses to a specific value (such as by picking *i*=42).

Unfortunately, in our experiments with dozens of exploitable bugs we found concretization overconstrains formulas, leading our initial AEG techniques to miss 40% of known exploitable bugs in our test suite;[12] for example, AEG may need to craft an input that becomes valid shellcode after being processed by `tolower` (such as `tolower` is *f* in Equation 2). In Mayhem, we proposed a number of optimizations for symbolic memory;[12] for example, one performs a type of strength reduction where symbolic memory accesses are encoded as piecewise linear equations.[12]

Example application: Exploiting `/usr/bin`. Recall we fuzzed Debian `/usr/bin` and found 52 distinct bugs in 29 programs, including `acpi_listen`. One goal was to determine which bugs are exploitable.

We ran our binary-only AEG tool called Mayhem[12] on each crash to determine if we could automatically generate an exploit from the crashing path. We also manually checked whether it was possible to exploit the bug. Five of the 52 bugs were vulnerable to a control-flow hijack, and Mayhem generated exploits for four of them. The exploit for `acpi_listen` took 0.5 seconds to generate, and the remaining three took 8, 12, and 28 seconds, respectively.

Automatically generating an exploit provides proof that the reported bug is security-critical.

These results on `/usr/bin` offer three insights: First, current AEG tools like Mayhem are sound but incomplete. A sound AEG technique says a bug is exploitable if it really is exploitable, while a complete technique reports all exploitable bugs. Unfortunately, Rice's theorem implies developing a sound and complete analysis for any nontrivial program property is in general undecidable. Second, AEG can be very fast when it succeeds. And finally, there is ample room for improving AEG in particular and symbolic execution and software model checking in general. For example, we analyzed why Mayhem failed on the last vulnerability, finding the problem was a single constraint that pushed the SMT solver (we use Z3) off an exponential cliff. Perhaps comically, manual analysis showed the constraint was superfluous but was not recognized as such by the automatic formula optimizer. Once the constraint was removed from the formula, exploit generation took less than five seconds.

Real-World Considerations

Security practitioners often focus only on exploits for programs on the attack surface of a system.[27] The attack surface consists roughly of the set of programs, files, protocols, services, and other channels available to an attacker; examples include network daemons, programs called from Web servers on untrusted inputs, privileged programs, and media players. Our example `acpi_listen` is not on the attack surface. We chose `acpi_listen` because it highlights the steps of AEG, yet disclosing the exploit would do little damage because it is not on the attack surface. Interestingly, the `acpi_listen` vulnerability is remarkably similar to a recent PHP vulnerability that performs an unchecked copy on the same data structure.[14]

Overall, AEG techniques are valuable because they show whether a pro-

gram can be exploited regardless of whether it is on the attack surface or not. For example, a program not on the attack surface in one deployment may be on the surface for another. More generally, programs on the attack surface are simply a subset of all programs; if we can handle all programs we can surely handle the subset on the attack surface. Current techniques have found exploits on the attack surface, albeit not in widely used large applications like Internet Explorer. For example, as we wrote this article we ran Mayhem on additional examples that are on the attack surface, finding a number of zero-day exploits for media applications (such as `ezstream` and `imview`) and network applications (such as `latd` and `ndtpd`).

Another consideration is additional layers of defense that might protect otherwise exploitable programs. Two popular operating-system-level defenses against control-flow hijacks are data-execution prevention, or DEP, and address space layout randomization, or ASLR.

DEP marks memory pages either "writable" or "executable" but forbids a memory page from being both. DEP prevents an exploit that requires writing and then executing shellcode on a memory page from working (such as the shellcode mentioned earlier). Unfortunately, attackers have developed techniques to bypass DEP. One such method is called return-to-libc where the attacker shellcode executes code already present in memory (such as by running system ("`/bin/sh`") in `libc` directly) rather than writing new code to memory. Return-oriented programming, or ROP, uses instruction sequences already present in memory, called "gadgets." Shacham et al.[33] showed it is possible to find a Turing-complete set of gadgets in `libc`.

ASLR prevents control-flow hijacks by randomizing the location of objects in memory. Recall that to exploit `acpi_listen`, the attacker needs to know the address of the shellcode. ASLR randomizes addresses so vulnerable programs likely crash instead of successfully redirecting control to the shellcode. ASLR is an important defense but does not fix the underlying vulnerabilities and thus may provide limited protection; for example, Windows and Linux systems running on 32b processors may have insufficient randomness to provide strong security,[34] though 64b architectures can address this problem. Particular deployments of ASLR may have weaknesses as well; for example as of January 2013 the program image of Linux executables is often not randomized. Even when randomized well, additional vulnerabilities may disclose information that can subsequently be used in a control-hijack exploit.

Schwartz et al.[31] proposed exploit hardening, which takes an exploit that works against an undefended system and hardens it to bypass defenses. One step in exploit hardening is to automatically generate ROP payloads (to bypass DEP) that take advantage of small portions of unrandomized memory (to bypass ASLR on the 2013 implementations of ASLR on Windows 7 and Linux). In particular, Schwartz et al. showed ROP payloads can be generated for most programs in Windows and Linux that have at least 20KB of unrandomized code, which is true for many programs. Exploit hardening can be paired with AEG to check the end-to-end security of a program running on a specific system.

Finally, DEP and ASLR defend only against memory overwrite attacks. Other vulnerabilities (such as information disclosure, denial of service, and command injection) are also critical in practice; for example, DEP and ASLR do not protect against exploits for the command-injection vulnerability found by Mayhem in `ezstream`.

Conclusion

AEG is far from being solved. Scalability will always be an open and interesting problem. As of February 2013, AEG tools typically scale to finding buffer overflow exploits in programs the size of common Linux utilities. In Mayhem, one current bottleneck is driving the symbolic executor to the buggy portion of the state space. As a result, programs with deep bugs are typically beyond the scope of our current Mayhem AEG tool. Examples include large programs with bugs deep in the program (such as Internet Explorer and Adobe Reader), as well as those with large protocol state (such as first authenticate, then send multiple fragmented messages to exploit a bug). In addition, programs with complex functions (such as hashes) are often a bottleneck for SMT solvers. One promising data point is that Microsoft's SAGE tool routinely finds bugs in large applications,[20] though automatically generating exploits for those bugs is an open challenge with huge potential rewards.

More fundamentally, AEG must expand to involve a wider variety of exploitability properties and scale to new program domains. While buffer overflows continue to be exploited[3,35] integer overflows, use-after-free, heap overflows, and Web vulnerabilities are also important (and popular) targets;[3] for example, heap overflows against modern operating systems like Windows 8 pose difficult challenges (such as modeling internal heap metadata and new heap allocators with built-in defenses). In our experience, the problem is often not coming up with some formalism, but with the right formalism and optimizations that make AEG efficient and practical on real-world programs and vulnerabilities.

Except for a few examples (such as Ganapathy et al.'s[17] exploits against a particular cryptographic API), most work in AEG has focused on exploiting programs in type-unsafe languages, though type safety is no panacea. Information flow, command injection, and many other common exploitable bugs can all occur in typical type-safe languages. Moreover, the runtime environment itself may have security-critical flaws. For example, the most commonly exploited vulnerabilities in 2011 were in Java.[2]

AEG can be modeled as a verification task; therefore, the better programmers and researchers get at software verification, the better they will likely get at automatically generating exploits. Some security researchers are pessimistic about the practicality of AEG in many application settings,[36] rightfully pointing out significant scalability hurdles and the lack of exploits against vulnerabilities like use-after-free. We are more optimistic. Eight years ago, AEG techniques were restricted to analyzing a single API call. Today, AEG can both automatically find and generate exploits in common binaries. In an effort to improve

security in Debian, we started a project in 2013 to check all programs in `/usr/bin` for exploitable bugs and so far have found more than 13,000 with more than 150 exploitable. Advancements will continue to be fueled by better tools, techniques, and improvements in verification and security.

Acknowledgments

This research is partially supported by grants and support from the National Science Foundation, the Defense Advanced Research Projects Agency, Lockheed Martin, Northrop Grumman, and the Prabhu and Poonam Goel Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Avgerinos, T., Cha, S.K., Lim, B.T.H., and Brumley, D. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium* (San Diego, CA, Feb. 6–9). Internet Society, Reston, VA, 2011, 283–300.
2. Batchelder, D., Bawany, S., Blackbird, J., Blakemore, E., Faulhaber, J., Fayyaz, S., Felstead, D., Henry, P., Goel, N.K., Jones, J., Kuo, J., Lauricella, M., Malcolmson, K., Ng, N., Oram, M., Peccelj, D., Probert, D., Rains, T., Simorjay, F., Stewart, H., Thomlinson, M., Wu, S., and Zink, T. *Microsoft Security Intelligence Report 12* (July–Dec. 2011). Microsoft, Redmond, WA; http://www.microsoft.com/security/sir/archive/default.aspx
3. Bilge, L. and Dumitras, T. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the ACM Conference on Computer and Communications Security* (Raleigh, NC, Oct. 16–18). ACM Press, New York, 2012, 833–844.
4. Bounimova, E., Godefroid, P., and Molnar, D. *Billions and Billions of Constraints: Whitebox Fuzz Testing in Production.* Technical Report MSR-TR-2012-55. Microsoft, Redmond, WA, May 2012; http://research.microsoft.com/apps/pubs/?id=165861
5. Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT—A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, Apr). ACM Press, New York, 1975, 234–245.
6. Brumley, D. and Jager, I. *Efficient Directionless Weakest Preconditions.* Technical Report CMU-CyLab-10-002. Carnegie Mellon University, Pittsburgh, PA, July 14, 2010; https://www.cylab.cmu.edu/research/techreports/2010/tr_cylab10002.html
7. Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E.J. BAP: A binary analysis platform. In *Proceedings of the International Conference on Computer Aided Verification* (Snowbird, UT, July 14–20). Springer, Berlin, Heidelberg, Germany, 2011, 463–469.
8. Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy* (San Francisco, May 18–21). IEEE Press, Los Alamitos, CA, 2008, 143–157.
9. Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation* (San Diego, CA, Dec. 8–10). USENIX Association, Berkeley, CA, 2008, 209–224.
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., and Engler, D.R. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security* (Alexandria, VA, Oct. 30–Nov. 3). ACM Press, New York, 2006, 322–335.
11. Cadar, C. and Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 2 (Feb 2013), 82–90.
12. Cha, S.K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing Mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy* (San Francisco, May 21–23). IEEE Press, Los Alamitos, CA, 2012, 380–394.
13. Chipounov, V., Kuznetsov, V., and Candea, G. The S2E platform. *ACM Transactions on Computer Systems 30*, 1 (Feb. 2012).
14. CERT/NIST. *PHP socket_connect() Stack Buffer Overflow. National Vulnerability Database,* Entry *CVE-2011-1938.* National Institute of Standards and Technology, Gaithersburg, MD, May 31, 2011; http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1938
15. De Moura, L. and Bjørner, N. Satisfiability Modulo Theories: Introduction and applications. *Commun. ACM 54*, 9 (Sept. 2011), 69–77.
16. Flanagan, C. and Saxe, J.B. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (London, U.K., Jan. 17–19). ACM Press, New York, 2001, 193–205.
17. Ganapathy, V., Seshia, S.A., Jha, S., Reps, T.W., and Bryant, R.E. Automatic discovery of API-level exploits. In *Proceedings of the International Conference on Software Engineering* (St. Louis, MO, May 15–21). IEEE Press, Los Alamitos, CA, 2005, 312–321.
18. Godefroid, P. Compositional dynamic test generation. In *Proceedings of the ACM Symposium on the Principles of Programming Languages* (Nice, France, Jan. 17–19). ACM Press, New York, 2007, 47–54.
19. Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Chicago, June 12–15). ACM Press, New York, 2005, 213–223.
20. Godefroid, P., Levin, M.Y., and Molnar, D. SAGE: Whitebox fuzzing for security. *Commun. ACM 55*, 3 (Mar. 2012), 40–44.
21. Grenier, L. (Pusscat and Lin0xx). Byakugan: Automating exploitation. In *ToorCon Seattle* (Seattle, WA, May 2007); http://seattle.toorcon.net/
22. Heelan, S. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities.* M.Sc. thesis. University of Oxford, Oxford, U.K., Sept. 3, 2009; http://solo.bodleian.ox.ac.uk/primo_library/libweb/action/dlDisplay.do?vid=OXVU1&docId=oxfaleph017069721
23. Howden, W.E. Methodology for the generation of program test data. *IEEE Transactions on Computers C-24*, 5 (May 1975), 554–560.
24. Jhala, R. and Majumdar, R. Software model checking. *ACM Computing Surveys 41*, 4 (Oct. 2009).
25. King, J.C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.
26. Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Beijing, June 11–16). ACM Press, New York, 2012, 193–204.
27. Manadhata, P.K. and Wing, J.M. An attack surface metric. *IEEE Transactions on Software Engineering 37*, 3 (May–June). IEEE Press, Los Alamitos, CA, 2011, 371–386.
28. Medeiros, J. *Automated Exploit Development, The Future of Exploitation Is Here.* Technical Report. Grayscale Research, 2007; http://www.grayscale-research.org/new/pdfs/toorcon_whitepaper.pdf
29. Muller, T. *ASLR Smack & Laugh Reference Seminar on Advanced Exploitation Techniques.* Technical Report. RWTH Aachen University, Aachen, Germany, Feb. 2008.
30. Saxena, P., Poosankam, P., McCamant, S., and Song, D. Loop-extended symbolic execution on binary programs. In *Proceedings of the International Symposium on Software Testing and Analysis* (Chicago, July 19–23). ACM Press, New York, 2009, 225–236.
31. Schwartz, E.J., Avgerinos, T., and Brumley, D.Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium* (San Francisco, Aug. 8–12). USENIX Association, Berkeley, CA, 2011, 379–394.
32. Sen, K., Marinov, D., and Agha, G. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering* (St. Petersburg, Russia, Aug. 18–26). ACM Press, New York, 2005, 263–272.
33. Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security* (Alexandria, VA, Oct. 29–Nov. 2). ACM Press, New York, 2007, 552–561.
34. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security* (Washington, D.C., Oct. 25–29). ACM Press, New York, 2004, 298–307.
35. van der Veen, V., dutt-Sharma, N., Cavallaro, L., and Bos, H. Memory errors: The past, the present, and the future. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses* (Amsterdam, The Netherlands, Sept. 12–14). Springer, Berlin, Heidelberg, Germany, 2012, 86–106.
36. Vanegue, J., Heelan, S., and Rolles, R. SMT solvers for software security. In *Proceedings of the USENIX Workshop on Offensive Technologies* (Bellevue, WA, Aug. 6–7). USENIX Association, Berkeley, CA, 2012.
37. Wang, X., Chen, H., Jia, Z., Zeldovich, N., and Kaashoek, M.F. Improving integer security for systems with KINT. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, Oct. 8–10). USENIX Association, Berkeley, CA, 2012, 163–177.

Thanassis Avgerinos (thanassis@cmu.edu) is a Ph.D. candidate in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA, and a founder of ForAllSecure.com.

Sang Kil Cha (sangkilc@cmu.edu) is a Ph.D. candidate in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA.

Alexandre Rebert (alexandre@cmu.edu) is a Ph.D. student in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA, and a founder of ForAllSecure.com.

Edward J. Schwartz (edmcman@cmu.edu) is a Ph.D. candidate in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA.

Maverick Woo (pooh@cmu.edu) is a systems scientist in CyLab at Carnegie Mellon University, Pittsburgh, PA.

David Brumley (dbrumley@cmu.edu) is an assistant professor in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA, and CEO of ForAllSecure.com.