Editors: Dino A. Dai Zovi, ddz@theta44.org
Alexander Sotirov, alex@sotirov.net

Vulnerability Detection Systems

Think Cyborg, Not Robot

Sean Heelan
Immunity Inc.

Has academic research improved the state of software security? In more abstract disciplines, questions of real-world impact can be irrelevant or difficult to answer, but not so for research aimed at vulnerability detection and related areas. In these areas, researchers typically aren't claiming to live the life of the mind but are striving for practical results backed by solid theory. An obvious test to determine whether this research has become relevant is to look at the techniques and tools people use to find vulnerabilities and understand software. Through this lens, the last two decades of software-verification and static-analysis research have unfortunately had negligible practical impact. The vast majority of publicly disclosed vulnerabilities are still found by fuzzing or manual auditing.

So why is the security community shunning systems based on academic research in favor of seemingly primitive methods? The answer is simple: results. Those who professionally find bugs and take apart software care about results, and the prototypes demonstrated by academics generally don't provide sufficiently useful results to motivate industrial interest.

Here I discuss why academic research has failed to create effective vulnerability detection software and offer some suggestions on how we can reap practical benefits from future research. The reasons for this failure also help explain why this software can't be completely automatic but must incorporate human knowledge and capabilities to be effective.

The Problems with Prototypes

Often, the goal of developing prototypes of vulnerability detection software appears to be complete automation. This goal usually isn't feasible, so we end up with systems that tackle parodies of real problems.[1] When professionals test these prototypes in real-world scenarios, the outcome is typically underwhelming and frustrating. Such outcomes reinforce the stereotype that academic researchers ignore the true problems in favor of those that are more academically rewarding to write about.

For example, one of the most lauded projects in academic circles in recent years has been the University of California, Berkeley's BitBlaze project (http://bitblaze.cs.berkeley.edu). BitBlaze has produced papers discussing solutions to everything from automatic exploit generation to protocol reverse engineering to universal unpacking. Despite this, in real-world testing, the resulting prototypes haven't proven themselves any more useful than existing tools.[2]

This situation seems due to a disconnect between the problems researchers think are important and the actual problems that practitioners of vulnerability detection must face. This distorted view of reality results in poor prototypes that give an unflattering view of the theories and algorithms being researched.

Furthermore, many researchers' lack of familiarity with hands-on bug finding, reverse engineering, and exploit development means that sometimes the problems receiving the most attention don't require it. Conversely, we're missing out on important research opportunities as researchers overlook nonobvious but important problems. Even when the issues are correctly identified, the resulting theories can be underdeveloped in critical areas necessary for real-world application. In short, this research has had limited impact because researchers focus on the wrong problems, and even when the correct problems are identified, the wrong applications are developed to demonstrate the results.

All of this of course raises the question, why should we consider the practical applications of new research if the old techniques work well? Assuming we want to be more efficient and effective, we need improved tools. Vulnerability auditing is difficult, and humans are prone to mistakes and inconsistent work, and often have difficulty comprehending complex code. If we look at how most people understand code and

 1540-7993/11/$26.00 © 2011 IEEE

```
3629 } else if (val->unit == CSSParserValue::Function) {
3630   // There are two allowed functions: local() and format().
3631   CSSParserValueList* args = val->function->args.get();
3632   if (args && args->size() == 1) {
3633     if (equalIgnoringCase(val->function->name, "local(") && !expectComma) {
...
3636         CSSParserValue* a = args->current();
3637         uriValue.clear();
3638         parsedValue = CSSFontFaceSrcValue::createLocal(a->string);
```

Figure 1. A recent bug in the WebKit Web browser engine. At line 3638, the *string* type is accessed without a check on the *unit* variable of variable *a*. The overall effect is an information leakage vulnerability if *a* has a different type, because someone could fake the contents of the string structure, which contains a pointer and length attributes.

find vulnerabilities, we see a number of common tasks that appear partially automatable on the basis of techniques from model checking, symbolic execution, abstract interpretation, and other popular abstraction and reasoning approaches. Examples of these tasks include finding a given input to reach a program location, checking a code base to see whether a function is ever called under certain conditions, finding ranges for variables, and checking for known bad idioms. Academic research could provide relevant input, but it must change. The tools we build to assist these tasks must work in the real world, and they must fit into the workflows of those who will use them.

Bad Demonstrations Kill Good Research

One of the few program-analysis approaches to gain popularity outside academic circles is symbolic execution. Based on successful research into techniques for solving satisfiability modulo theories (SMT), symbolic execution has laid the groundwork for an impressive number of papers and tools over the past few years. The ability to model code and make queries about it using a decision procedure is incredibly powerful and could be useful in reverse-engineering software.

However, most symbolic-execution systems are useless for real code in real scenarios. Consider Klee, probably the best-known public tool for finding bugs through symbolic execution. Unless Klee is dealing with a particularly well-behaved application, it's lucky to explore more than a fraction of a program's states in any reasonable runtime. This significantly curtails its bug-finding abilities, and as a tool for integrating into a human code auditor's workflow, it's generally of little use.

Often, auditors need the ability to apply symbolic execution to code deep within an application. Typically, the auditor will have contextual information on function arguments, global variables, and local variables and will be able to inform a symbolic-execution system on a useful portion of the program's state. In this context, the potential problems of interest differ from those of whole-program analysis. Selecting paths to explore automatically from input points is no longer as much of an issue. The focus instead becomes how best to include and use information provided by a user and how to intuitively communicate analysis results.

If we think about symbolic execution this way, bug finding can be built up from localized model checking that's integrated into an auditors' workflow to the more ambitious projects we see today that look at whole-program analysis. The difference between the two ends of the spectrum is that the supposedly less impressive prototypes that demonstrate analysis at a smaller scale can actually be useful.

For example, Figure 1 demonstrates a recent bug in WebKit (www.webkit.org), an open source Web browser engine. The issue is at line 3638. The variable `a` contains a union that's either a `string` (containing a `pointer` and `length`), `double`, `int`, or `pointer` structure. It also contains a `unit` variable that indicates the type of `CSSParserValue` and hence what member of the union to access. At 3638, the `string` type is accessed without a check on the `unit` variable of `a`. The overall effect is an information leakage vulnerability if `a` has a different type, because someone could fake the contents of the string structure, which contains a pointer and length attributes.

On discovering such a bug, we often want to search the rest of the code for similar mistakes. That is, we want to search for accesses to the union variables of `CSSParserValue` that aren't preceded by a check of `unit` on that same structure. This task could be incredibly simple or difficult, depending on the code. We could

generalize this problem to one of annotation or property checking, which is an ongoing research area at several places, including Microsoft's Havoc project (http://research.microsoft.com/en-us/projects/havoc). Or, we could take a different direction and investigate how best to integrate static type checking, dataflow analysis, and other techniques.

Academic researchers in security have an obligation to educate themselves properly on modern security bugs, vulnerability detection methods, and exploitation techniques. The price for not doing so is irrelevance.

The point is this: on the basis of current SMT solving and static-analysis technology, problems such as the one I just described are within our grasp. Some tricky issues require further research to solve, but much of the work has been done. Perhaps problems such as finding the best way to make symbolic execution human-guidable and useful in real workflows don't seem academically glamorous. However, this attitude must change if we're to see prototypes that can have a real impact and do justice to the other research they draw upon.

Gaps in Our Knowledge

Even more troublesome than applying useful theories and algorithms in ways that fail to demonstrate their usefulness is overlooking important research areas or solving them incompletely. That leads to gaps in our research output that are glossed over or simply ignored.

One such case of this involves the threats that client applications such as Web browsers face. Most discovered bugs with a severe impact are in the use-after-free class. These bugs are prolific and often easier to exploit than a traditional stack or heap overflow. However, a review of modern academic security research might lead someone to conclude that the threat landscape hasn't changed in nearly a decade. Most of this research is stuck on outdated benchmarks and searching for overflows that will just as likely be found with a fuzzer. The research that does test with real programs tends to focus on unknown or unmaintained software of little practical or intellectual interest.

Considering the research effort that has gone into finding buffer overflows and the fact that we still don't have useful tools, perhaps when we look at use-after-free bugs, an opportunity exists to develop our theories in a different direction.

Current automatic methods are unsuccessful for modern applications. There are several reasons for this, one of which is that most prototypes struggle to deal with C++ idioms such as inheritance, virtual functions, templates, and function overloading. They also have difficulty with applications that have event-based control flow, which can mean that paths containing the use of a freed pointer might not directly follow from those on which it is freed.

The task of analyzing Web browsers with embedded JavaScript engines illustrates this problem well. The bugs that plague these applications tend to involve a called JavaScript function that manipulates the browser's document object model (DOM) such that an object is freed in the background but a pointer to it remains as an attribute of another object in the DOM. The use of this variable is then triggered through another JavaScript function call or direct attribute access. If you're analyzing the underlying C/C++ code without knowledge of the JavaScript engine, this bug will slip by. Trying to solve this problem entirely automatically is difficult. Perhaps it's feasible with a precise model of the effects of every JavaScript function call, but building that in itself is a formidable problem.

A more tractable approach in this case is to trace the propagation of pointers and then detect locations that still might contain stale references after a `free` function. Many systems already include this activity. However, it doesn't seem to be valued as a result itself, so few researchers have evaluated how capable we are at this specific task. On real C++ code, the answer is simply that we're incredibly bad at even basic pointer tracing. Even less research has been done into how best to present the results of a stage such as this to a human auditor, so the output is useful neither programmatically nor directly by a human.

Focusing on performing this single activity for real-world C++ code and doing it well is more worthwhile than trying to attack the entire problem at once and doing it poorly. Determining how to access these stale pointers can then be treated as a separate problem either manually by the user or automatically. This seems more natural for two reasons. First, the algorithms for both stages of this activity (tracing pointer propagation and detecting locations with stale references) will differ considerably. Second, it makes sense to strive for human-usable output from both stages.

If we accept the opinion that academic research often focuses on the wrong problems and, even

more so, fails to demonstrate results in a useful and compelling manner, where do we go next?

First, we must move away from the idea that end-to-end, entirely automated solutions are feasible demonstrations of research results and are what we should aim for. My reasoning for this is twofold. First, the prototypes built so far have done the opposite of showing the practical effectiveness of a collection of theories and algorithms. Research teams typically don't have the budget or expertise to complete such projects in a manner that produces a functioning tool.

Second, significant, under-investigated potential exists for static-analysis techniques designed for integration into a human code auditor's workflow. Complex issues arise when we consider applying the current state-of-the-art static-analysis research in this context. A human expert will typically have significant amounts of internalized knowledge, and integrating it with algorithms is nontrivial. The potential payoff is worth it, though, if we can use human-provided information to deal with situations in which automated algorithms often struggle—that is, analysis of loops, dynamic allocation, function pointers, compiled applications lacking type information, and so forth. Microsoft's Havoc project and other recent research seem to confirm this approach's usefulness.

Academic researchers in security have an obligation to educate themselves properly on modern security bugs, vulnerability detection methods, and exploitation techniques. The price for not doing so is irrelevance. Industry researchers, for their part, sometimes impulsively dismiss academic solutions. If we're to improve our tools, which are at times incredibly primitive, we must get beyond this and reflect on how we work and how we can use mathematical abstractions and algorithms to become more productive and effective. □

Acknowledgments

Thanks to Rolf Rolles, Julien Vanegue, and Alex Sotirov for providing valuable feedback during the drafting of this article. The phrase "think Cyborg, not robot," comes from a Tim O'Reilly discussion on artificial intelligence.

References

1. T. Avgerinos et al., "AEG—Automatic Exploit Generation," *Proc. 2011 Network and Distributed System Security Symp.* (NDSS 11), Internet Soc., 2011; http://security.ece.cmu.edu/aeg/aeg-current.pdf.
2. C. Miller et al., "Crash Analysis with BitBlaze," 2010; http://securityevaluators.com/files/papers/CrashAnalysis.pdf.

Sean Heelan *is a security researcher with Immunity Inc. Contact him at seanheelan@gmail.com.*