Kestrel

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304
Ph/Fx: (650)493-6871

Network Vulnerability Analysis

A Formal Approach

James McDonald
John Anton

March 14, 2001

Contents

1 Background **1**
- 1.1 Principles . . . 1
- 1.2 Tools . . . 2
  - 1.2.1 Specware . . . 2
  - 1.2.2 MetaSlang . . . 2
  - 1.2.3 Designware . . . 3
  - 1.2.4 Java Byte Code Verifier . . . 3

2 Generic Approach **3**
- 2.1 Language . . . 3
- 2.2 Analysis Tools . . . 4
- 2.3 Taxonomy of Flaws . . . 4
- 2.4 Mixed Initiative . . . 4
- 2.5 Overview . . . 5

3 Example **6**

4 Specific Approach **10**
- 4.1 Shadowable Methods . . . 10
- 4.2 Sensitive Regions . . . 10
- 4.3 Refinement of Taxonomy Node . . . 11

5 Results **12**

6 Future Work **13**

Abstract

This note describes early, but promising, results using Kestrel's technology to address the problem of locating real vulnerabilities in commercial off-the-shelf (COTS) software. We provide the rationale for our approach, give a brief description of it, document some early results, and list our current view of the next development steps.

1 Background

We begin with a summary of the underlying technology to show that our approach is firmly grounded in systematic engineering principles.

1.1 Principles

A few basic principles have guided the development of *Specware*^{TM}, and much of the success of this project can be attributed to following those guidelines.

First and foremost is the principle of ***synthesis from design***, or alternatively, ***synthesis from specifications*** or ***synthesis by refinement***: code is generated by correctness-preserving refinements or transformations from abstract specifications, to provide ***correctness-by-construction*** in the final code. In what follows, specifications may also be referred to as specs, while the term *theory* is used to refer to the deductive closure of a specification: specs are the finite presentations of infinite theories.

The second principle is ***orthogonality***: specifications of different aspects of an application are designed, developed, and maintained as independently as possible from each other. Some theories describe a domain such as scheduling or graph layout, while others describe data structures such as strings or hash tables, while yet others describe algorithmic strategies such as divide-and-conquer or global-search. To date, Specware has focused on functional specifications that just define what a program should do, but potentially it could encompass theories to describe architectural aspects such as networking or client-server arrangements, computational aspects such as data latency or memory performance, or perhaps even algorithmic complexity and other meta-properties.

Third is the principle of ***semantic modularization***: specifications of any kind are designed as many small theories with strong semantic connections to each other. Each small theory is large enough to encompass the concepts and constraints comprising some meaningful software component, but is otherwise as small as possible, to maximize reusability. The connections used to glue theories together are required to maintain strict semantic compatibility, which can be thought of as behavioral type-checking (or even more informally as type-checking with a vengeance).

Fourth is the principle of ***taxonomies***: theories and techniques tend to form natural hierarchies from abstract to detailed versions, and such information can be captured and presented to developers as guides for software development. Furthermore, the structure of such a taxonomy may be exploited when progressing from designs using one node in the taxonomy to those using another node, especially if the second is a specialization of the first.

Fifth is the principle of ***automation***: as much as possible of the development process should be automated, to keep the developer's attention focused on just the crucial specification and design issues.

1.2 Tools

1.2.1 Specware

Specware^{TM} is the generic system developed at Kestrel to formally compose many small algebraic specifications into a complete specification of some problem, and to then refine that specification down to executable code. [22] [23]

Various techniques from category theory [3] [12] [13] [17] [24] are used for generic composition and refinement operations, while the component algebraic specifications themselves are implemented in a rather simple language called Slang.

1.2.2 MetaSlang

MetaSlang is a new language intended to replace Slang, with a type system that is simpler for users. Additionally, many tools for manipulating MetaSlang and the categorical machinery have been written in MetaSlang itself, so it is largely bootstrapped.

1.2.3 Designware

Designware is a level of organization layered on top of Specware to facilitate the use of taxonomies, enable complex operations to be invoked via scripts, perform simplifications and other optimizations on specs and code, etc. [19] [20] [21]

1.2.4 Java Byte Code Verifier

Kestrel's Java Byte Code Verifier was specified in MetaSlang for the purposes of another project. It embodies the first formal specification by anyone of a Java byte code verifier, and the very process of specifying it revealed inconsistencies within the informal specification of such a verifier and flaws in Sun's implementation that allowed un-type-safe code to pass inspection.[6] [9] [18]

2 Generic Approach

Applying the framework above to the task of analyzing COTS [1] software developed by others is not immediately obvious, but there is a viable strategy that begins with a semantically explicit taxonomy of vulnerabilities at the top-level juxtaposed against semantic representations of the target software at the bottom level, and which uses an opportunistic mixed initiative approach to construct a refinement of some node in the taxonomy down into theories located within the representations of the target program.

To succeed, this approach needs several components: languages, tools, taxonomies, and an analysis environment.

2.1 Language

Whatever technology is employed, in a mature tool of the type we envision it should be possible to readily express essential notions related to vulnerabilities. This would require semantic theories, both for abstract terms such as resource, agent, time, etc. and for concrete terms such as Unix's "fread" system call, Netscape's "SecurityManager" Java class, etc. which are grounded in the semantics of particular programming languages, operating systems,

[1] commercial off-the-shelf

network protocols, virtual machines, actual processors, etc. In general, such a language would also be expressive enough to describe modalities such as time, knowledge, necessity, etc.

Constructing such a full language would be a multi-year effort, far beyond the scope of this project, but we have created small pieces of it driven by the needs of describing particular vulnerabilities. Our expectation is that each new example considered will augment the language available for use in subsequent examples.

2.2 Analysis Tools

Since in general we are analyzing programs for which we may not even have source code, let alone documentation or specifications, we will need a suite of analysis tools that can create detailed concrete semantic representations of such programs. In general, such tools will create formal, explicit representations of the kinds of structural knowledge used implicitly by traditional compilers and decompilers, plus ad hoc knowledge about the semantics of various class libraries such as security managers, resource allocators, etc.

2.3 Taxonomy of Flaws

Much of the ultimate power of this approach will come from the codified knowledge embedded in semantic taxonomies of flaws. Many taxonomies of vulnerabilities have been published (e.g. [1], [2], [4], [5], [7], [8], and [11]), but as far as we know, ours will be the first to create formal semantic connections among the nodes in such a taxonomy. Given an embedding of one vulnerability node into a target, the arrows in the taxonomy contain enough semantic information to constrain the embeddings of related nodes for formal consistency.

This allows the user to find an embedding of a detailed vulnerability in the target program by proceeding through successively more elaborate descriptions of the vulnerability, one small step at a time, each step prompted by the results of the previous step within a mixed-initiative framework.

2.4 Mixed Initiative

In general, the problems being solved are far beyond the capabilities of fully automated tools–it will be crucial to provide an environment in which knowledgeable humans will be able to make strategic choices and guide the exploration of alternatives.

2.5 Overview

The diagram in figure 1 provides a graphical view of how all these pieces might fit together in a mature vulnerabilities analysis tool.

$$\begin{tikzcd}
\begin{array}{c}\text{Abstract Semantics of}\\ \text{Generic Vulnerability}\end{array} \arrow[dr, "from\ taxonomy"] \arrow[dddd, Rightarrow, "initial\ simple\ refinement"'] & & & \\
& \cdots \arrow[dr, "from\ taxonomy"] \arrow[dddl, Rightarrow, "extension"] & & \\
& & \cdots \arrow[dr, "from\ taxonomy"] \arrow[ddll, Rightarrow, "extension"] & \\
& & & \begin{array}{c}\text{Abstract Semantics of}\\ \text{Particular Vulnerability}\end{array} \arrow[dlll, Rightarrow, "final\ extended\ refinement"] \\
\begin{array}{c}\text{Concrete Semantics}\\ \text{of COTS Target}\end{array} & & & \\
COTS\ Target \arrow[u, dotted, "analysis\ tools"] & & &
\end{tikzcd}$$

(1)

Figure 1: Overview of Process

First the analysis tools would be run to get specifications for the concrete semantics of the COTS artifact.

Second, the user would pick a generic node in the taxonomy and attempt to refine the abstract terms within it down to terms in the concrete semantics, such that all axioms for the abstract terms translate into concrete claims provable within the specifications of the concrete semantics. In essence this shows that an instance of that abstract generic theory can be found within the particular semantics of the COTS target.

Then, iteratively, the user would pick successively more particular nodes in the taxonomy, at each step extending previous results to construct a refinement of the current node into the concrete semantics.

3 Example

To keep this project grounded in reality, we chose to study particular significant vulnerabilities reported in the literature, with the goal of creating the minimal tools within the framework above that would suffice to locate instances of some abstraction of the chosen vulnerability. Conceptually, the chosen abstraction would be a node in the envisioned taxonomy of vulnerabilities.

To this end, we focused our attention on the "Brown Orifice" attack on Netscape Navigator developed by Dan Brumleve and demonstrated at http://www.brumleve.com/BrownOrifice/. This vulnerability is documented by CERT as "CA-2000-15: Netscape Allows Java Applets to Read Protected Resources" at http://www.cert.org/advisories/CA-2000-15.html, and with bugtraq ID 1546: "Netscape Communicator URL Read Vulnerability" at http://www.securityfocus.com/bid/1546.

In essence, this attack downloads a "Trojan horse" applet that circumvents the security provided by Netscape's use of its capability classes, in such a way that the applet acts as a server exposing all of the victim's files to any inquirer on the net.

The attack uses the ability of a Java subclass to shadow methods from its parent class to create evil instances of Sockets and ServerSockets that behave in unanticipated ways within code designed to handle security related to sockets.

In particular:

- The "close" method of class Socket was shadowed by a "close" method that does nothing, confounding attempts by ServerSocket to close a socket that was tentatively opened but then found to be in violation of the security policy.

  In figure 2, we see that the close method for instances of the library Socket class will invoke impl.close, but the close method for instances of the evil BOSocket class instead does nothing.

Thus in figure 3, we can see that if implAccept is passed a normal Socket, the close method invoked in the catch of a SecurityException calls impl.close, but if implAccept is passed a bogus BOSocket, the invoked close does nothing.

```
public class Socket {
  ...
  public synchronized void close()     <== NORMAL METHOD
    throws IOException
   { impl.close(); }                   <== NORMAL ACTION
  ...
}

public class BOSocket extends Socket {
  public void close_real() throws IOException
   { super.close(); }
  public void close()                  <== SHADOWING METHOD
   { }                                 <== EVIL NON-ACTION
}
```

Figure 2: Shadowable library class Socket, and an evil Subclass

- In figure 4 we see a variant of accept (called accept_any) that has been added to a subclass of the library class ServerSocket. This method thwarts the attempt by implAccept (shown above), which it calls, to throw a security exception out to the context invoking accept_any.
- Finally, in figure 5, we see that the constructor method for class URL-Connection is shadowed by a method that sets the protected variable "connected" to true, no matter what problems may been have detected.

```
public class ServerSocket {
  ...
  protected final void implAccept(Socket socket)
    throws IOException
  { try
    { socket.impl.address = new InetAddress();
      socket.impl.fd = new FileDescriptor();
      impl.accept(socket.impl);
      SecurityManager securitymanager =
         System.getSecurityManager();
      if(securitymanager != null)
      { securitymanager.checkAccept
            (socket.getInetAddress().getHostAddress(),
             socket.getPort());
        return; }}
    catch(IOException ioexception)
    { socket.close();
      throw ioexception; }
    catch(SecurityException securityexception)
    { socket.close();                 <== SHADOWABLE
      throw securityexception;        <== INTERCEPTABLE
    }}
  ...
}
```

Figure 3: Library class ServerSocket with a sensitive region

```
public class BOServerSocket extends ServerSocket {
  ...
  public BOSocket accept_any()    <== NEW METHOD
    throws IOException
  { BOSocket s = new BOSocket();
    try { implAccept(s); }        <== MAY THROW EXCEPTION
    catch (SecurityException se)  <== INTERCEPTION
      { }                         <== EVIL NON-ACTION
    return s;
  }
}
```

Figure 4: Method that intercepts security exception

```
public class BOURLConnection extends URLConnection {

  public BOURLConnection(String u)  <== SHADOWING CONSTRUCTOR
     throws MalformedURLException
  { super(new URL(u));
    connected = true; }             <== EVIL ACTION

  public BOURLConnection(URL u)     <== SHADOWING CONSTRUCTOR
  { super(u);
    connected = true; }             <== EVIL ACTION

}
```

Figure 5: Subclass of URLConnection that resets "connected"

All three tricks are interesting and can be generalized, but we focused on the first.

4 Specific Approach

The “Brown Orifice” trick with the “close” method of the “Socket” class exploits the intersection of two properties within Netscape system code: (1) a sensitive region in which issues related to security are being addressed, and (2) the invocation of methods for instances of a class that could in fact be shadowed by evil methods for instances of subclasses of that class.

Kestrel’s pre-existing Java Byte Code Verifier, written in MetaSlang, already had the ability to parse Java class files, construct a data flow representation of the code within each method, and use transfer functions associated with that representation to effectively type-check the byte code by solving a data flow problem expressed in terms of those transfer functions on a semilattice expressing the control-flow.

We exploited large portions of this verifier to extract two new abstractions: (1) invocations of non-final (i.e., shadowable) methods and (2) regions where security issues were being addressed.

4.1 Shadowable Methods

If a Java class is declared to be final, no subclasses are allowed. On a finer scale, if a Java class is not final, subclasses are allowed, but any methods within the class that are declared final cannot be shadowed in any subclass.

Thus there is a tradeoff between making generic classes that can be specialized by various subclasses, versus retaining control over methods that perform sensitive operations. This tradeoff introduces the opportunity for security mistakes, since a programmer may be unaware or negligent of some security concern while making such a tradeoff.

4.2 Sensitive Regions

In general, for our purposes a region of code may be considered sensitive for any of several reasons, but the essential notion is that some resource, perhaps an abstract resource such as a privilege, has been obtained for the duration of that region, or that the region is cleaning up the aftermath of some exception related to acquisition of a resource.

The pseudo-code in figure 6 illustrates the structure of some typical sensitive regions.

```
void Foo (LibraryClass C)
{ try
    { <acquire resource>;
      C.m1 }                          <== HIGHLY DANGEROUS
  catch(ImportantException exception)
    { C.m2 }                          <== MAY BE DANGEROUS
  finally
    { C.m3 }                          <== SUSPICIOUS
}
```

Figure 6: Pattern for various sensitive regions

For the particular experiment we have already run, sensitive regions were defined to be the catch clauses for exceptions of class SecurityException.

4.3 Refinement of Taxonomy Node

Given those two abstractions, we can define an abstract taxonomy node that says, in effect, "Find a sensitive region that invokes an unpredictable method". The refinement of this node can then be decomposed into the refinement of each of those notions: "sensitive region" and "unpredictable method". In turn, "sensitive region" can be refined to "sensitive syntax" and "sensitive semantics". We've seen three variations of syntax locations that are potentially sensitive, and the sensitive semantics might be inferred from calls to a security manager, resource allocator, alarm clock initiator, etc.

The fact that such a simple use is made of the taxonomy merely reflects the initial state of this project and our focus on a rather simply described vulnerability. In general, the taxonomy node might refer, for example, to several sensitive regions interacting in particular ways according to complex protocols. The refinement would then require more sophistication than merely finding an intersection of two features.

5 Results

We ran the resulting analysis tool on 1031 Java class files for Netscape that represent about 100,000 lines of source code, with the following goals:

1. Demonstrate that our generic approach finds the two specific needles (problematic invocations of a shadowable method exploited by "Brown Orifice") that we know are in the haystack: one invocation of a shadowable "close" method for instances of class "Socket" within a constructor method for ServerSocket and another such invocation within the implAccept method of ServerSocket.

2. Report no false positives – i.e. report only true vulnerabilities.

3. Find additional true positives – similar method invocations associated with new vulnerabilities.

4. Run relatively quickly, at a minimum for feasibility, and ideally fast enough to facilitate extensive experimentation.

How did we do? In short, pretty well.

1. We found the two specific "Brown Orifice" invocations we knew we needed to find.

2. We reported about five clusters of false positives, but these were due primarily to the use of an overly simple method for locating sensitive regions, and an overly verbose reporting mechanism, both easily correctable.

3. We found about four new suspicious invocations that might represent novel vulnerabilities, unreported to date.

4. The original unoptimized program was unacceptably slow, but one adhoc optimization allowed us to handle most of the 1031 files in about 10 hours of processor time. About 40 anomalous files were clearly not candidates for this vulnerability (they merely initialized large arrays of constants), but due to some missing optimizations[2] required enough

[2]Some operations on maps were behaving O(N**3) when O(N) is possible with a little care, and the maps in question were growing to size 6000 or so.

much processing time that we manually by-passed them. Relatively straightforward optimizations now underway for the overall process should provide substantial speed-ups.

6 Future Work

Current work is proceeding opportunistically on several fronts, but with the following general bias:

- First, to facilitate more rapid future experimentation, the existing program is being revised to make its component specifications simpler and more generically useful, and to optimize performance.
- Second, we are generalizing the notion of "sensitive region" to facilitate finding new classes of vulnerabilities. For example, code that has acquired a lock on a resource raises the spectre of deadlocks or denial of service.
- Third, we are analyzing the tantalizing results of new potential vulnerabilities reported in our initial look at Netscape Navigator, to see if in fact an attack can be constructed to exploit them.
- Fourth, we will analyze other known flaws in applications to see if we can find abstractions of vulnerabilities similar to the ones found for "Brown Orifice".
- Fifth, we will be applying the tools to other target COTS applications to see what other as-yet-undetected vulnerabilities we can find.
- Sixth, we will explore the development of new tools for extracting semantic information from COTS targets, to provide a richer target for refinement of abstract vulnerabilities.
- Seventh, we will begin to populate a formal taxonomy of vulnerabilities and build the tools needed to refine nodes in that taxonomy to abstracted specifications of COTS targets.

In general, we are quite pleased by promising initial results, and are excited by the extensive research paths we see to extend those results.

References

[1] Aslam, T. A Taxonomy of Security Faults in the Unix Operating System, MS thesis, Department of Computer Sciences, Purdue University, Coast TR 95-09, 1995.

[2] Aslam, T., Krsul, I., and Spafford, E. H. A Taxonomy of Security Faults, in *PROCEEDINGS OF THE NATIONAL COMPUTER SECURITY CONFERENCE*, Coast TR 96-05, 1996.

[3] Barr, M. and Wells, C. *Category Theory for Computing Science*, Prentice Hall, 1990.

[4] Bishop, M. and Bailey, D. A Critical Analysis of Vulnerability Taxonomies, in *Proceedings of the NIST Invitational Workshop on Vulnerabilities* (July 1996)

[5] Bishop, M. A Taxonomy of UNIX System and Network Vulnerabilities, ECS-95-10, Department of Computer Science, University of California at Davis (Sep. 1995)

[6] Coglio, A., Goldberg, A., and Qian, Z. Towards a Provably-Correct Implementation of the JVM Bytecode Verifier In *Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java*, Vancouver, B.C., October 1998. Technical report KES.U.98.5, Kestrel Institute, Palo Alto, CA., August 1998.

[7] Cohen, F., Phillips, C., Swiler, L. P., Gaylor, T., Leary, P., Rupley, F., Isler, R., and Dart, E. A Preliminary Classification Scheme for Information System Threats, Attacks, and Defenses; A Cause and Effect Model; and Some Analysis Based on That Model Sandia National Laboratories, September, 1998

[8] Du, W. and Mathur, A. P. Categorization of Software Errors that led to Security Breaches, in *21ST NATIONAL INFORMATION SYSTEMS SECURITY CONFERENCE*, CRYSTAL CITY, VA, 1998, Coast TR 97-09, 1997

[9] Goldberg, A. A Specification of Java Loading and Bytecode Verification Allen Goldberg. In *Proceedings, 5th ACM Conference on Computer and*

Communications Security, San Francisco, October 1998, Technical report KES.U.97.1, Kestrel Institute, Palo Alto, CA., December 1997.

[10] HUANG, X. A Comparison Between Standard and Formal Mathematical Software Development. Master of Science Thesis, University of Maryland Department of Nuclear Materials and Reliability Engineering, 1999.

[11] LANDWEHR, C. E., BULL, A. R., MCDERMOTT, J. P., AND CHOI, W. S. A Taxonomy of Computer Program Security Flaws, with Examples, In *ACM Computing Surveys*, Vol. 26, No. 3 (Sept. 1994), pp. 211-254

[12] LAWVERE, F. W. AND SCHANUEL, S. *Conceptual Mathematics: a First Introduction to Categories*, Cambridge University Press, 1997.

[13] MACLANE, S. *Categories for the Working Mathematician*, Springer Verlag, 1971.

[14] PAULK, M. C., CURIS, B., CHRISSIS, M. B., AND WEBER, C. V Capability Maturity Model, Version 1.1. In *IEEE Software, Vol. 10, Number 4*, IEEE Computer Society, Los Alamitos, CA., July 1993.

[15] PAVLOVIĆ, D. Semantics of first order parametric specifications. In *Formal Methods '99* (1999), J. Woodcock and J. Wing, Eds., Lecture Notes in Computer Science, Springer Verlag. to appear.

[16] PAVLOVIĆ, D. Compositionality via diagrams in software design. In progress.

[17] PIERCE, B. *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

[18] QIAN, Z. A Formal Specification of a Large Subset of Java(tm) Virtual Machine Instructions for Objects, Methods and Subroutines In *Formal Syntax and Semantics of Java(TM).* Alves-Foss, J. (Ed.), Springer Verlag LNCS, 1998. Technical report KES.U.98.4, Kestrel Institute, Palo Alto, CA., August 1998.

[19] SMITH, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[20] SMITH, D. R. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.

[21] SMITH, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.

[22] SRINIVAS, Y. V., AND JÜLLIG, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

[23] SRINIVAS, Y. V., AND MCDONALD, J. The Architecture of $SPECWARE^{TM}$, a Formal Software Development System. Technical report KES.U.96.7, Kestrel Institute, Palo Alto, CA., August 1996.

[24] TAYLOR, P. *Practical Foundations of Mathematics*, Cambridge University Press, Cambridge, U.K., 1999.