Fine-Grained Inspection for Higher-Assurance Software Security in Open Source

J. P. McDermott
Naval Research Laboratory
john.mcdermott@nrl.navy.mil

Abstract

This paper reports our experiences in using a fine-grained software inspection tool to increase productivity in both software security evidence construction and internals re-engineering, for a higher-assurance open source software project named Xenon. We explain why fined-grained tool-based inspection is essential for a higher-assurance open source project.

1. Introduction

Higher-assurance software is software that has been subjected to assurance engineering that exceeds conventional best practices for assurance. This assurance engineering not only reduces the number of residual security flaws below conventional best practice but also provides reviewable engineering evidence that this has been done. It is beyond the scope of this paper to address every assurance engineering practice. Instead, we address two key practices that, in some circumstances, require the use of a fine-grained software inspection tool: design decision capture and vulnerability analysis.

A design decision is essentially the intent, or “why”, of software source code, as opposed to the actual behavior, or “what”, of the code. For software to have higher-assurance security, design decisions must be captured and maintained with the source code, to a degree that exceeds conventional best practice. There are two reasons for this: 1) constructively adversarial design review and 2) security patching. Higher-assurance security requires not only source-code level vulnerability analysis but also adversarial review of the design to reduce flaws and increase confidence in the software. Capture of design decisions must also exceed conventional best practice, to increase the likelihood that security patching is possible and that patching flaws will not introduce new flaws. If design decisions are not captured, then a security patch may be designed in a way that contradicts a lost design decision; thus the patch will not actually remove the flaw. If design decisions are not captured, then a security patch may violate a lost design assumption or principle and introduce a new security flaw.

Some design decisions are captured directly by the source code. For example, the actual modules of a software product are frequently (but not always) captured in this way. (As an example of where this is not the case, the Xen source code file xen/arch/x86/mm/shadow/multic contains 16 distinct modules.) Many other critical decisions are not captured by the source code. For a simple example, the secret of a module is not captured by the source code.

Open-source software is typically not higher-assurance and design decisions are not captured. If we have software that is a good candidate for higher assurance, we need a way to discover and capture that design information.

Automated vulnerability analysis tools do not apply to all software products. Some software products, by their nature, do not contain any of the vulnerabilities that automated vulnerability analysis tools detect. These products do contain vulnerabilities, but not the kind found in typical software, e.g. buffer overflows are not exploitable even if they exist. Two examples of this are hypervisors and biometric authentication software, as typified by the Xen open source hypervisor and the NSA-sponsored Tokeneer project, respectively.

Hypervisor security weaknesses are not typical of other software. After boot time, well-designed hypervisors get inputs via registers and fixed-size structures. Attacks based on overflows or commands embedded in strings are largely irrelevant. Nevertheless, hypervisors are not invulnerable. An alternative to automated vulnerability analysis is needed.

This paper explains how and why we used a fine-grained inspection tool to address these two issues in the Xenon project. The rest of this section describes the Xenon project, gives more details on higher-assurance...
open source, and discusses related work. The rest of the paper explains fine-grained software inspection, reports our results, discusses limitations, and presents conclusions.

1.1. Xenon

The Xenon project is investigating the construction of an open source separation hypervisor based on the open source Xen hypervisor [1]. According to Randell and Rushby, a separation kernel [2] is a “specialized operating system core . . . that focuses solely on the provision of isolated address spaces with controlled communications between them, while policy is enforced by trusted applications running in some of those address spaces.” A separation hypervisor is similar to but larger than a separation kernel; it provides stronger separation than a conventional hypervisor but, because of its larger size, is likely to have less assurance than a pure separation kernel. In exchange for this likely reduction in assurance, the separation hypervisor should have lower cost and better support for a range of commodity software running on a range of commodity hardware, e.g., the latest x86_64 instruction set and the full Advanced Programmable Interrupt Controller (APIC) standard.

It is also important to be clear that the Xenon prototypes are not “secure” versions of Xen, in the way that SE Linux relates to Linux. Just as the Xen open source hypervisor was initially developed from the open source Linux operating system, by simplifying Linux and modifying its design, NRL is developing the Xenon separation kernel from Xen. Figure 1 on the next page diagrams this, with rectangles depicting Linux code, and darker fill depicting greater assurance.

The Xen hypervisor design includes security features and simplicity that make it a good basis for an open source separation hypervisor. The Xen on source prototypes are open source and Xenon results will be made available to the Xen community. The first Xen prototype was based on version 3.1 of the Xen open source hypervisor. Currently, the Xen project is porting the first prototype to the 3.3 version of Xen, resulting in the second Xenon prototype.

The primary goal of the Xenon project is to investigate issues in re-engineering open-source software to higher security assurance than conventional open source software. Common Criteria [3] evaluation or some other form of certification [1] is not a goal, though our work is mindful of the Common Criteria.

Instead, we are investigating the direct approach [4] to higher-assurance security, that is, the construction of an assurance argument based on solving a specific security problem the Xen prototype claims to solve. Our direct approach uses many of the work products and practices found in the Common Criteria, but to greater or lesser extent than mandated by the higher evaluation assurance levels of the Common Criteria. The direct approach uses formal methods, internals simplification, specific security policies, and significant documentation beyond the source code itself. In contrast to the Common Criteria, we make greater use of formal methods, and provide less informal documentation.

Our interest in the direct approach is that 1) it may reduce certification costs; and 2) it is probably essential for higher-assurance open-source. Our interest in re-engineering conventional open source into higher-assurance form is that it should offer the same benefits as conventional open source, but be suitable for applications requiring higher assurance. Specific applications that require higher assurance may also require specific certifications that differ from application to application. For example, Common Criteria evaluations at Evaluation Assurance Levels [3] above EAL 4 are not recognized internationally, so even within a single EAL of the Common Criteria there is need for different approaches. Developing to a specific certification scheme or set of criteria probably reduces the size of the community that would be interested in supporting the software.

This paper reports our experiences in using a fine-grained software inspection tool to increase productivity in both evidence construction and internals re-engineering, for higher-assurance open source code. We explain why this productivity is essential for a higher-assurance open source project.

1.2. Higher-Assurance Open Source

The key characteristic of higher-assurance software is that, by some means, we have greater confidence that the software satisfies some specific properties [4]. Properties are a more precisely defined instance of “intent”: what the software is supposed to do as opposed to what it actually does.

Most open source does not record or even define intent. The working assumption is that cases where the code does not match the intent are so infrequent and obvious that it is not cost effective to define or record intent. Best practice conventional open source assumes that there is a small group of developers who do understand the intent of the code; they do not need to record or analyze it. The design decisions are not captured but held in the minds of the responsible experts. In many circumstances, this is an effective approach. However,
this approach does not support higher-assurance.

Higher-assurance open source code must define, record, and analyze intent, even when there is a group of developers who understand the intent. There are three reasons for this: 1) even infrequent issues resulting from a mismatch between intent and the code must be addressed, 2) the intent of the higher-assurance code may be different (changed), and 3) for security, intentions in general and design intentions in particular must be subjected to a constructively adversarial third-party review ([5], Chapter 26).

When we re-engineer conventional open source software to higher-assurance, we typically will not have the assistance of the developers who already have deep understanding of the source code. The goals of conventional open source are different. Conventional open source projects cannot divert scarce developer resources to explaining intent or assisting in evidence construction. So a higher-assurance open source project needs to be efficient in acquiring expertise without the assistance of those who already have it.

More significantly, higher-assurance security specialists in a higher-assurance open source project will have to acquire deep understanding of larger amounts of code than typically supported by open source developers. There will be fewer of them, so they will be expected to cover more code. Also, since attacks can involve exceptional usage of software features, a security specialist may need to acquire deep understanding of code not involved in typical use cases.

1.3. Related Work

Because it is related to high-assurance host platform security, the Multiple Independent Levels of Security (MILS) work [6, 7] is related to the Xenon project. MILS encompasses several projects, for example the LynxSecure project [8], but they all differ from Xenon because they are focused on achieving Common Criteria evaluation.

Because the Xenon project has a goal of providing an open example of high assurance, it is related to the Naval Postgraduate School’s Trusted Computing Exemplar Project [9]. The Trusted Exemplar Project is focused on the highest level of assurance and is not based on re-engineering existing open source.

The NSA-sponsored Tokeneer project [10] provides a complete open example of a high-assurance software product with extensive use of formal methods. Tokeneer is a special purpose software product of approximately 9000 non-comment lines of source code.

Toll et al. [11] report on the tools they used to build a high-assurance smart card operating system at the IBM Thomas J. Watson Research Center. They faced a similar issue that smart card vulnerabilities are typically different from the kinds of vulnerabilities addressed by analysis tools with built-in definitions. In contrast, their work was not involved with re-engineering but with ab initio construction of high-assurance software, so recovery of undocumented intent was not an issue for their work.

2. Tool-Based Fine-Grained Program Inspection

To achieve the necessary productivity in the Xenon project, we need tools that can provide general direct deep understanding.

General understanding means that the tool is not designed or implemented with specific predefined understanding. For example, a security vulnerability analysis tool has built in definitions of what it means to be a “security vulnerability”. The tool provides direct deep understanding, but only of security vulnerabilities as it defines them; so the understanding is not general. If we are interested in a new class of vulnerability not built
in to the tool, then we cannot use the tool to gain deep understanding.

Direct understanding means: 1) that no annotations, assertions, or other supporting work products have to be supplied to the tool, and 2) the source code does not have to be translated from the target language to some other language or formalism. Requiring either of these implies the user already has some deep understanding of the source code.

We use the term deep understanding to mean something more than examination of surface-level syntactic features, e.g. searching for strings in the source text. Examples of work products for deep understanding include not only formal specification, formal verification, model checking, program annotation, pseudocode, and natural language specifications but also desk checking[12], abstract syntax trees, program dependence graphs [13], and program slicing [14].

Formal annotations and specifications are important approaches to increasing the security assurance of software; the Xenon project uses formal specifications. Arguably, both annotations and specifications provide deeper forms of understanding. However, one must be in a position to use them. Deep understanding is required before either can be constructed, so neither can be used to provide the initial deep understanding.

There are at least two practical situations where we are not likely to be in a position to use either annotations or specifications. First, as is the case with the Xenon project, it may be that the developers who already have deep understanding of the code chose not to supply annotations or specifications. As we have explained above, the developers may have good reasons for not doing this. The second practical situation is where annotations or specifications have been constructed, but for different properties than the ones we are interested in. An obvious example would be software that had formal annotations or specifications for flight safety, when our interest is in security against human adversaries. So tools for direct deep understanding can have relatively wide applicability even when tools for annotation, formal verification, or automatic vulnerability analysis are also available.

Because of its general direct deep understanding capabilities and applicability to a C/Unix/Xen environment, the Xenon project chose CodeSurfer as its fine-grained inspection tool. We also chose CodeSurfer because there is no alternative tool. Other available static analysis tools that apparently support general direct deep understanding either are not general (because the tool uses built-in definitions of what is a problem) or are not direct (because they require construction of some kind of annotation or supporting work product).

We do not list these tools because we do not want to create the false impression that any of them is generally unsatisfactory. The other tools simply are not general deep direct understanding tools.

CodeSurfer analyzes both C and C++ and also runs on Windows, but we did not need C++ or Windows code understanding. P. Anderson et al. [15] provide the best detailed explanation of the CodeSurfer code inspection tool.

We summarize briefly here that CodeSurfer analyzes the build of a target software product and constructs a cross-referenced database of dependencies, flows, pointer relationships, calls, and variable use information. There are five major components of the interface. Figure 2 shows a screen shot of all of them, with a Property Sheet in the center, and clockwise from the upper left, the File Viewer, Project Viewer, a Call Graph Viewer, and the Finder.

- **Project Viewer**: Users can see the file and function structure of a build, in hierarchical form. From the Project Viewer, users can invoke the File Viewer, Call Graph Viewer, or Property Sheet Viewer. They can also pose program slicing queries. A backward slice of a point P in a program shows 1) all the points that can influence whether control reaches point P and 2) all the points that can influence the values of variables used at point P. There are also forward slice and chop queries. Points in the slice are marked in both the Project Viewer and in any instance of a File Viewer that applies, so users can see both the fine-grained local part of a slice and also the global project-wide view. In Figure 2, the tick marks on the right side of the Project Viewer are showing the results of a slice based on the global variable frame_table. (Other aspects of the slice are shown in all of the windows, but are not clear in the figure because they are based on colored syntax highlighting.)

- **File Viewer**: Users can navigate, read, copy, and query (but not modify) the source code. File viewer queries include slicing, call graphs, and invocation of Property Sheet Viewers. Navigation includes jumping to declarations, type definitions, macro definitions, and program points.

- **Call Graph Viewer**: Users can build, edit, and save call graphs. Any C function can be added to (or removed from) a call graph and all of its callers and callees can be automatically added or deleted. The user can change the layout of the call graph by dragging functions to new locations; connections are automatically updated. Users can also jump to a File Viewer for any C function in the call graph.
3. Tool-Based Inspection Experience in the Xenon Project

We now present some specific experiences of our use of fine-grained code inspection in re-engineering the Xen conventional hypervisor into the Xenon separation hypervisor. We do not explain the entire direct assurance approach used for the Xen project, but only those activities or work products that benefit directly from tool-supported fine-grained code inspection.

3.1. Hypervisor Weakness Analysis

The current emphasis in the Xen project is on removing security weaknesses rather than discovering security vulnerabilities. A design or code construction choice that could be exploited as all or part of a vulnerability is a weakness. The distinction is that we may not know if a weakness is a vulnerability. Instead, we only know that it has the potential to be a vulnerability. Since the Xen project is focused on security, weak design or code construction choices may be changed to stronger approaches, even if there is no confirmed exploit.

The idea of removing weaknesses is similar to the concept of code smells [16] used in agile programming.

- **Property Sheet**: Users can see or query a variety of properties for a selected file, function, data structure, or variable. One of the most important for the Xen project is variable use. Variable use queries include defs where variables are changed, uses where variables are read, and cond-kills where variables might be changed. For the Xen project, another useful query displayed by a Property Sheet is the list of compiler flags actually used to build a selected file or function. The Xen makefiles are complex and the meaning of some parts of the Xen source code is heavily dependent on the applicable makefile, so it is helpful to have tool-based analysis of the effects of these files.

- **Finder**: Users can use the Finder to search for functions, variables, structs, unions, typedefs, pointers, and plain old text. The results of a Finder query can be selected for investigation via a File Viewer, Property Sheet, or Call Graph Viewer.
A code smell is not necessarily a problem, but indicates that further investigation should be made towards refactoring the code.

When removing the weakness is relatively inexpensive, we expend the engineering resources on removing it instead of confirming that it is a vulnerability. When removing the weakness is potentially difficult, we may choose to confirm that it is a vulnerability, by constructing an exploit.

Hypervisors are not invulnerable, but the weaknesses they do have are not typical of other software. After boot time, the Xen hypervisor gets its inputs via registers and fixed-size structures in designated hypercall pages; Xenon uses the same hypercall mechanism. Attacks based on overflows or commands embedded in strings are largely irrelevant for the hypervisor itself. (The only string inputs are taken from local configuration files used to start the hypervisor or one of its guests. An attacker who is in a position to modify these is already in control of the hypervisor, so input string attacks would be pointless.)

The significance of this is that most automatic vulnerability analysis tools are less important for hypervisor weakness analysis, because they are not focused on typical hypervisor vulnerabilities.

We have found that desk checking [12] is more useful for hypervisor weakness analysis than it is for other kinds of software. Desk checking is the practice of selecting test data (inputs) and manually executing the software for the test. Desk checking differs from program reading because the control flow and corresponding program states are explicitly recorded, typically in a tabular format. Desk checking provides a permanent record of what was read or inspected, line by line. It also provides the same contextual information that a documented test case provides, such as the purpose of the desk check and basis for selection of input values.

Hyphervisor testing is relatively expensive, in terms of the scaffolding that must be constructed and maintained. Desk checking requires no scaffolding but is just as useful, in the analysis of corner cases and hypothetical attacks, for discovering weaknesses.

Extensive desk checking of complex hypervisor source code could be impractical without a fine-grained inspection tool. The Xen source code is well-structured; concerns are separated to a very fine granularity. Typically, a reader or desk checker would have to visit 3 to 5 other files to see all of the definitions and logic related to a single line of code in a single desk check. Without a fine-grained inspection tool, identifying the other files and the specific lines that apply to the line of code in question typically takes several minutes. Using a fine-grained inspection tool, a desk checker can typically visit the related code as though browsing a series of web sites. The difference may not seem significant for a single desk check, but minutes-per-line is impractical when inspecting or desk checking a large number of cases.

3.2. Formal Specification

Formal methods are an essential part of higher-assurance software. In the Xenon project, we have defined a formal security policy [17] and we are constructing a formal specification of the hypervisor interface and its internal design. The formal security policy model uses the Circus formalism [18]. The interface and internal design specifications are being constructed in the Z formalism, using Standard Z [19] and the Community Z Tools (CZT) framework [20]. Construction includes a bottom-up approach that builds Z state space and operation schemas directly from the source code.

Deep understanding of the Xen source code is vital for this kind of modeling. The Xen community has produced good high-level documentation and explanation of the API, in books [21], publications, and white papers. The next-level details of the internal architecture are almost entirely undocumented.

When constructing a Z schema from Xen source code, the specifier must have a deep understanding of the state space and operations of interest. She or he must also be able to discern which code construction choices in the source determine intent and which code construction choices are implementation decisions. Implementation decisions should not find their way into the Z specifications.

The “surfing” aspect of the inspection tool is essential for constructing these Z specifications. A user constructing Z specifications of Xenon must simultaneously have a clear understanding of the intended meaning of the state space and operations represented by the C/assembly source code and the state space and operations represented by the Z notation. Without such a tool, the user will experience significant digressions from the Z construction activity, in order to answer fine-grained questions about the intended meaning of the C/assembly source.

These digressions are more expensive during formal modeling. During a digression, the specifier loses focus not only of the intent of the C source code but also the intent of the Z specification. A tool that minimizes this loss of focus is critical.
3.3. Attack Space Reduction

The Xenon project defines the attack space size of a software product as the size of its input state space. More specifically, we use the input state space size $W_d$ defined by Brady, Anderson, and Ball [22] in their model of software reliability. R. Anderson provides an example of the adverse consequences of attack space size for a plausible large attack space. In the example, the defender has 4 orders of magnitude more security test and analysis resources than the attacker, but only has a 1% chance of finding the same security flaw as the attacker [5], because of the large attack space size. This is in spite of the fact that the defender discovers 4 orders of magnitude more flaws than the attacker.

The Xenon approach to attack space reduction is removal of code. The code is not removed because there is a problem with it. Instead, the code that is judged least likely to be used in applications of Xenon is removed, to reduce attack space size. We remove code on a feature-by-feature basis rather than fine-grained line-by-line removal. We consider the code removal to be successful if the resulting prototype still supports both Windows XP and OpenSUSE Linux as unmodified (i.e., HVM) guest operating systems. Fine-grained inspection allows us to determine if removing code from one part of the code base will impact the Xenon prototype’s ability to support these guest operating systems.

Removal of features entails a kind of requirements analysis at a level above the code details. Clearly, conditional code for configurations not supported by the Xenon prototype can be removed. This would include support for the Intel Itanium processor. We also remove code that supports features we do not expect to be used in Xenon. Examples of these features would be Non-Uniform Memory Access (NUMA), the fast reboot, and live kernel replacement capabilities of Xen’s kexec feature. This high-level “requirements” analysis does not benefit directly from fine-grained inspection. Previous information resulting from other inspection may be useful, but this is not direct support by inspection. Once a feature has been targeted for removal, fine-grained inspection becomes essential.

Once code has been identified for removal, the inspection tool is used to locate all of the program points that must be modified to delete the code. Code removal is done in two phases. First, a slicing-based feasibility analysis gives us a detailed estimate of the amount of code that is associated with a target feature, and how dependent the rest of Xen is on that feature. If the removal is feasible, then further slicing is used to identify all the program points that need to be removed.

We apply slicing by using the tool to slice on the program points that implement a feature. If a forward slice from a program point $p_1$ that is targeted for removal includes code from another program point $p_2$, then $p_2$ could fail to work if program point $p_1$ is no longer present. A series of slices will provide an informal “map” of the dependencies that apply to a feature of the hypervisor.

If the feasibility analysis shows us a well-structured modular relationship between a feature’s implementation and the rest of the code then the feature is a feasible candidate for removal. Further slicing and also variable usage in a Property Sheet (defs, cond-kills and uses) are used to quickly locate the lines of code that are affected by a selected piece of Xen source. The CodeSurfer tool we use also shows unused conditional code directly, by means of syntax highlighting. This highlighted unused code is a “low-hanging fruit” that we also exploit, but it is not the same as the code we identify by slicing.

Attack space reduction in the Xenon project depends not only on the efficiency introduced by the fine-grained inspection tool but also on the high modularity of Xen code. If the Xen code were not modular, it would be difficult to remove performance sensitive features like NUMA.

A fine-grained code inspection tool is essential for removing large amounts of code, with the resources likely to be available to a higher-assurance open source software project. In our experience with Xenon, this has been more certain and faster than using text searching and build error messages.

Removing code to reduce the attack space has the added benefit of reducing the size of the assurance argument. Code that is not present does not need formal specifications, vulnerability analysis, assurance argument cases, or any other form of evidence.

3.4. Include File Analysis

In order to separate concerns and increase modularity, the Xen source code has a complex include file structure. There are multiple include files with the same name, distinguished by their paths. For example, there are six include files named xen.h. The Xen source code also has complexity in transitive file inclusion. An include file may contain its own include commands, with these containing further include commands. The semantics of each inclusion can be conditional; some inclusions have no effect. It is difficult to determine the effect of all the files that are being include, from reading the Xen source code.

In the Xenon project, we have requirements to both modify and simplify the Xen include file structure. Making strong design and code construction choices
During this re-engineering requires deep understanding of the include file structure. Prior to using the inspection tool, we used a combination of ctags, doxygen, dot, and custom Ruby scripts. As an alternative, the inspection tool shows unused include files directly in its File viewer. That is, in its display of a given file, the File viewer shows the include commands with different syntax highlighting, if the include files were not used in the current build. For example, in the latest build of the second Xenon prototype, we can see that include files include/xen/config.h, include/public/xen.h, and include/public/domctl.h are not used in header file include/xen/sched.h. (This is not a problem with Xen, but a natural consequence of our build configuration.) So, by direct inspection, we understand that we can modify those include files without having an effect on include/xen/sched.h. Clearly, we can also remove the include commands from file include/xen/sched.h with no effect.

The inspection tool Property Sheet for an include file will also show us transitive includes, all of the files that are included via the given include file. This makes it relatively inexpensive to understand the consequences of moving C header code, e.g. an inline function definition, from one include file to another.

We have used the inspection tool to 1) create new include files based on a modified include file structure, 2) move code from one include file to another, and 3) remove include commands from specific C language files. None of these activities requires a sophisticated software inspection tool, but the alternatives are much more time-consuming. Calculating the actual effect of file inclusion manually, without the inspection tool, requires not only inspection of the C source code files but also understanding of the configuration and make files used to build Xen.

3.5. Encapsulation

Higher-assurance software must make less use of global variables. Non-constant global variables used as parameters significantly increase the scope of the functions that use them. They make code hard to read and increase the chance of introducing a bug when changing the code. Non-constant global variables also impact thread safety in an adverse way. On the other hand, there is no evidence, i.e. experimental result, that prohibiting global variables has greater assurance than severely restricted use of them.

One of the sub-goals of the Xenon project is to investigate the actual impact of global variables on higher-assurance. We are not prohibiting the use of global variables per se, but investigating the actual assurance impact of allowing the justified use of some global variables. Every global variable that is used is explicitly justified, so there will be no future questions as to why the variable is global. The low-level design choice to encapsulate or not requires a significant amount of fine-grained information. In open-source software, or any software that comes without detailed documentation of intent, on-the-fly tool-based inspection of the use of a global variable is essential.

A typical example of tool-based re-engineering of Xen global variables into Xenon is the analysis of the global variable structpage_info*frame_table. Global variable frame_table holds the starting address of the frame table, as determined during the Xen boot process. It is defined in the source file arch/x86/mm.c and set to the correct value by function init_frametable (also defined in mm.c) by a call from function __start_xen (defined in arch/x86/setup.c). Variable frame_table is read by Xen code that needs to know where the frame table starts, for example in converting the address of a page information structure into a page frame number. As we will see shortly, global variable frame_table is not used to pass parameters, but acts as a constant whose value can only be determined at run-time.

We would like to understand all uses of frame_table, to decide if and how it should be encapsulated or re-factored in some other manner. A single CodeSurfer Finder query will show us that there is only 1 declaration (i.e. in arch/x86/mm.c), 381 uses, 2 defs, and 3 cond-kills of frame_table. (The interested reader is encouraged to try to find all 381 uses of frame_table, by means of conventional text searching.)

Use of the Finder and its the source code navigation feature shows us that the defs are 1) the compiler initialization and 2) the call to function init_frametable (also defined in mm.c) from __start_xen, i.e. frame_table is set at boot time and only read after that, with the exception of the the 3 cond-kills. A further Finder single-click query shows us directly that all three cond-kills are via C function vsnprintf. We can use the navigation feature on the Property Sheet for vsnprintf to see that all 3 cond-kills happen in C source file common/kexec.c. Given the usage of kexec in Xen (approximately the same as for Linux kernels, i.e. to replace the running kernel with a different kernel), it is a reasonable and “safe” redefinition of the frame table starting address. However, the Xenon project is not likely to retain kexec functions in the final separation hypervisor prototype. We also learn, from...
the uses, that frame_table is employed in many address calculations of the page and frame management functions of Xen. Requiring a function call for each of these uses would impose a performance penalty.

If we choose to encapsulate frame_table, we could apply the uses and defs discovered by the Finder to analyze the access function requirements. Each use would have to be replaced with an appropriate access function call, e.g. “get_frame_table_starting_address()”. Each def would have to be replaced with an appropriate “set_frame_table_starting_address()” access function call. Based on our experience, this replacement can be non-trivial if the original access to the global is a complex indirect access. A typical example of this kind of complex access would be use of the Linux kernel container_of macro that takes a pointer to a member of a structure and returns a pointer to the structure that contains it.

3.6. Simplification

Like the Common Criteria ADV_INT family of assurance requirements, our direct approach minimizes internal complexity. Simplification (i.e. reducing the complexity of the code) not only makes code easier to read and less likely to contain security flaws but also can work to reduce attack space $W_0$. The Xen project takes no position on the merits of any particular complexity metric. There are many competing definitions for complexity metrics, discussion of them is beyond the scope of this paper.

We are currently using cyclomatic complexity [23] because it is widely understood, computed automatically the inspection tool, and easy to compute by hand (for contributors who don’t have a tool). The Xen project limits the complexity of any C function to be less than 20. Our practice on the Xen project is to either simplify code that exceeds the allowable value or to provide a documented justification for the complexity that is not removed. As we reported for the first prototype [24], most existing Xen code is well within this limit. The capability of a fine-grained inspection tool to “inspect” complexity at the same time as other source code properties is important for the Xen project. Because complexity can be inspected by the same tool that we use to inspect for security weaknesses, formal meaning, internal structure, or encapsulation, it is easy to consider complexity at the same time.

4. Current Limitations

The most apparent limitation in direct general deep-understanding tools is inherent. Tools for general deep understanding, as opposed to tools with specific built-in definitions of understanding, have no expertise encoded within. The generality of the tool requires expertise from the user. Direct general deep-understanding tools make the expert user much more efficient, so they reduce costs by reducing the amount of expert time needed. However, direct general deep-understanding tools do not reduce the amount of expertise required.

We do not consider this to be a serious limitation. Direct general deep-understanding tools apply best to situations where “built-in” expertise does not apply, so a high level of expertise will be required always.

Direct general deep-understanding tools complement automated vulnerability analysis. In applications of higher-assurance software, an attacker is likely to use some form of fine-grained inspection, tool assisted or otherwise, to find vulnerabilities that are not discovered by automated vulnerability analysis. The defenders must also work in this space, to provide some corresponding protection.

Currently there are no direct general deep-understanding tools that provide simultaneous inspection or deep understanding of combined assembler and C language source code. This is significant for the Xen project. Approximately 13% of the Xen code base is written in AT&T assembler, i.e. for GNU as. Out of the 510 source files that are inspected by the current tool, there are 22 C source or header files with inline assembler content. There are also 19 assembly language files with preprocessor content, i.e. Unix “.S files” and 1 raw assembly language file without preprocessor content, i.e. a Unix “.s file”. These 20 files are not inspected by the current tool. Variables, structures, and behavior critical to initialization, memory management, and control transfer are defined in this assembly language. The assembly language files (.s and .S files) are particularly troublesome because only the linker commands in the Makefile connect them to the C code. If inspection for security weaknesses, formal meaning, internal structure, or encapsulation traces through some assembler source code, the user is forced to fall back on less powerful tools and more time is consumed. This issue is currently under investigation in a separate research project.

5. Conclusions

Our experience in the Xen project is that tool-support for general direct deep understanding is essential to constructing open-source code for higher-assurance security. The practices we report here should apply to any open source code that is sufficiently mod-
ular and simple. Re-engineering conventional open source code into higher assurance form involves several activities where general direct deep understanding is required. These activities include

- discovering and removing security weakness that are not amenable to automated vulnerability analysis;
- capturing the design decisions or intent of the source code, in formal specifications;
- reducing the attack space; and
- improving the internal structure.

The key developers of the conventional open source, who already have deep understanding, cannot provide on-demand fine-grained exposition of their generally undocumented code. Tool-support for general direct deep understanding allows third-party high-assurance security specialists to compensate for this.

A more general conclusion is that tool-supported fine-grained inspection increases the scope of direct general deep understanding by higher-assurance security specialists. This is so, even if the code is not third-party open source, but code produced by the project itself. It applies even when re-engineering is not the goal.

References


