Abstract

The importance of software assurance is growing, but traditional development techniques have not kept pace with this need. In critical domains such as utilities and defense systems, the shift in technology from hardware-centric, disconnected devices to software-centric, connected devices has exposed many vulnerabilities in existing systems. The trend toward an Internet of Things with everything connected to everything else compounds the problem. Using traditional code-centric software development techniques and relying on source scanners to find vulnerabilities in the code is doing too little too late.

This white paper discusses advances in system development that represent a major step forward in increasing software assurance. While no technique is applicable in every situation and software assurance is only one part of an overall cybersecurity plan, the security and resilience of both newly developed systems and legacy systems can be significantly increased.

Introduction

There are a number of capabilities, techniques, tools, and concerns that fall under the umbrella of “cybersecurity”. In this white paper, we will focus on the area of software assurance for both the development of new software and the hardening of legacy software.

The available taxonomies of software vulnerabilities, such as the Common Weaknesses Enumeration, tend to be specific to programming languages or attack patterns. Addressing these concerns is a component of a comprehensive software assurance plan, but it is not in itself sufficient. To get a broader view, this paper addresses three main questions related to software assurance:

- Do the requirements for the system capture what is intended?
- Does the developed system do what was specified in the requirements?
- How can one protect existing software?

There are other aspects of software development related to cybersecurity. For example, it is important to ensure that software libraries linked to an application are free from malicious code and that safeguards are in place to protect the build servers from physical attack. These issues, which largely relate to the environment in which the software was developed and currently operates, are beyond the scope of
this paper. We will concentrate here on how the software was developed and currently operates and on the characteristics of the developed code.

Software Assurance

A good working definition of software assurance is from the Department of Homeland Security. They describe software assurance as being comprised of three major parts: Trustworthiness (the lack of exploitable vulnerabilities, either maliciously or unintentionally inserted), Predictable Execution (justifiable confidence that software, when executed, functions as intended), and Conformance (a planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements and standards/procedures). Other working definitions can be used, such as the “CIA Triad” of Confidentiality, Integrity, and Availability. However, the details of what is included are not as important as their intended effect on the deployed systems.

For each of the three main questions above related to software assurance, there are related questions that should be asked to get a deeper understanding of the issues involved and how they may be addressed. The rest of this section discusses these related questions.

Do the requirements for the system capture what is intended?

This main question has two aspects to it: do the requirements specify the correct system, and do the requirements specify the system correctly? Collectively, the people who want the system to be built have in mind what they want the system to do (although there is frequently some disagreement on the details). Going from this “mental picture” of the software to a comprehensive set of requirements is fraught with error. First, the natural language (e.g., English) used to write the requirements is inherently ambiguous. Second, the person writing the requirements will leave out some detail that is assumed to be obvious to the reader. Third, the person writing the requirements may accidently leave out some requirements, make some mistakes, or introduce additional ambiguity. Fourth, the “mental picture” of the requirements may not actually provide all the properties that are intended.

To address these issues, we will group them into three sub-questions. The first one is whether or not the requirements as stated match what the creator of the requirements had in mind. That is, the written requirements must match not only the mental picture held by the creator of the requirements, but they must also build that same mental picture in the reader. Requirements that are written in a natural language are good for building a mental picture, but their ambiguity and lack of structure make it nearly impossible to ensure that it is the correct mental picture. On the other hand, requirements that are specified in a fully formal language leave no room for ambiguity of meaning, but it is typically extremely difficult to build a mental picture from them. Any solution to this issue must find a balance between natural expression and rigor.

The second sub-question is whether the requirements as stated are both complete and self-consistent. That is, independent of what exactly the system is supposed to do, are there basic problems with the structure of the requirements which indicate that they cannot possible be correct. If there is any state that the system be in (except the final state) such that the system can make no further progress, the requirements are incomplete. This indicates that requirements have been missed. For example, requirements would be incomplete if they state that an error condition must be logged, but not how the system should recover from that error. If there is any state that the system can get to in which the system can do multiple things with no way to decide which one to do, the requirements are inconsistent. This indicates that multiple conflicting (or overlapping) requirements exist for a given situation. For example, requirements would be inconsistent if one requirement states that if there are no free channels a channel request is denied, and another requirements states that if there are no free channels a channel request is
retried until it is granted. A solution to this issue should be able to automatically find such problems in the requirements.

The third sub-question is whether the requirements as stated have any safety or liveness issues. That is, are the important application-specific properties guaranteed to be true? Safety and liveness properties are a means of providing information about the system separate from the requirements. Safety properties should always be true. For example, a safety property may specify that no unregistered device may access the network. Liveness properties will eventually become true in some state. For example, a liveness property may specify that a reset command will cause the network to go back to its initial state. Safety and liveness properties are a means of providing information about the system separate from the requirements. A solution to this issue should be able to automatically prove or disprove (with a counterexample) such properties about the requirements.

**Does the developed system do what was specified in the requirements?**

Even if the requirements are perfectly specified and understood, that is not a guarantee that there were no issues introduced during the rest of the development process. The largest contributing factor to implementation issues is human error. Even industry-leading practices such as those found in CMMI Level 5 organizations still lead to defect rates of approximately 0.9 defects per thousand lines of code. Traditionally, an organization tries to remove these defects through a combination of practices: design reviews, code reviews, additional testing, etc. However, these practices have two major drawbacks. First, the practices are fallible because they are performed by humans. Second, they generally attempt to catch defects after they are already introduced instead of stopping them from happening in the first place. An effective solution to this issue should minimize the opportunities for human error.

We are not implying that reviews and testing should not be done. Rather, it is important to remove the source of human errors in the development process. For example, to the extent possible, a comprehensive test suite should be automatically created directly from the requirements, and this test suite should be used to automatically test both the design and the implementation.

A second, and perhaps more damaging, source of implementation issues is the surreptitious insertion of malicious code. The larger and more complex a system is, the easier it is to hide backdoor code or other malicious functionality. A solution to this issue should provide the appropriate level of transparency to the design and implementation.

**How can one protect existing software?**

Given the inevitability of defects introduced by humans in a system and the ever-growing list of attack vectors against the system, one cannot assume that any development method is sufficient in itself. Defensive cybersecurity must be an active process after deployment. A solution to this issue should address how one can minimize the damage of a successful attack on an existing system.

Trying to protect deployed software is made more difficult with legacy code. Proper software assurance techniques were often not used in the original development of the code. Rewriting these systems is typically prohibitively expensive, and patching the code, when that is even an option, often leads to the introduction of additional defects. A solution to this issue should address how legacy systems can be made more secure in a cost-effective manner.

**Solutions to Software Assurance Issues**

No one technology or technique can address all of the software assurance concerns. However, the consistent application of a number of solutions can provide a comprehensive framework for software
assurance. The table below summarizes the software assurance issues discussed above and lists solutions that can be applied. The solutions are discussed in the remainder of this section.

<table>
<thead>
<tr>
<th>Software Assurance Issue</th>
<th>Potential Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do the requirements as stated match what the creator of the requirements had in mind?</td>
<td>Provide a high-level and easy-to-review notation for capturing the complete use cases of the system.</td>
</tr>
<tr>
<td>Are the requirements as stated complete and self-consistent?</td>
<td>Provide a high-level and easy-to-review notation for capturing the complete use cases of the system.</td>
</tr>
<tr>
<td></td>
<td>Perform formal requirements verification.</td>
</tr>
<tr>
<td>Do the requirements as stated have any safety or liveness issues (i.e., are the necessary application-specific properties guaranteed to be true)?</td>
<td>Perform formal requirements verification.</td>
</tr>
<tr>
<td>How can one minimize the opportunities for errors in development?</td>
<td>Use automation where possible to minimize human errors.</td>
</tr>
<tr>
<td>How can one have confidence that the system as built functions correctly?</td>
<td>Create and apply a comprehensive test suite guaranteed to cover all the requirements.</td>
</tr>
<tr>
<td>How can one ensure transparency of design and code such that malicious code cannot be inserted.</td>
<td>Ensure transparency of design and code such that malicious code cannot be inserted.</td>
</tr>
<tr>
<td>How can one minimize the damage of a successful attack on an existing system?</td>
<td>Ensure that the effects of compromised code are kept as local as possible.</td>
</tr>
<tr>
<td>How can one protect legacy code that was not originally developed to take security concerns into account?</td>
<td>Reengineer the legacy system to add modern software assurance practices.</td>
</tr>
<tr>
<td></td>
<td>Provide an application-aware security wrapper during the execution of the legacy code.</td>
</tr>
</tbody>
</table>

**Provide a high-level and easy-to-review notation for capturing the complete use cases of the system**

One powerful, high-level notation for capturing a complete set of use cases for a system is Use Case Maps (UCMs). UCMs are standardized in ITU-T Recommendation Z.151. The UCM notation allows the creation of scenario paths through components involved in the scenario. These paths describe causal relationships between responsibilities of one or more use cases, showing related use cases in a map-like diagram. An example UCM is shown in the figure below.
UCMs have several advantages related to software assurance. The formal nature of the notation itself means that many inconsistencies and incompletenesses in the requirements will be caught as the UCM is being created. The notation captures much information in a compact and easily understood form, thereby reducing the cognitive burden on the analyst for complex systems. UCMs also integrate many scenarios into a unified and cohesive set of diagrams, which enables reasoning about potential undesirable interactions of scenarios. The UCM notation is simple and intuitive, leading to a low learning curve while providing an effective tool for requirements review.

**Perform formal requirements verification**

Finding defects in a system early in the development process not only leads to lower costs, but it also increases software assurance. It is not easy to consistently and completely correct defects after they have been introduced. Ideally, one would like to ensure that no defects are traced back to the requirements phase. One can have a higher level of confidence in the requirements for a system if properties can be proved (or disproved, for undesirable properties) about those requirements.

There are several different types of properties that can be proved. For example, an analyst would typically want to know that the requirements are complete and consistent before implementation begins. Consistency and completeness are generic properties in that they should be true of any system. The analyst may also want to prove that important application-specific or domain-specific properties of system are not violated. For these properties, an analyst uses safety and liveness conditions.

It is generally not sufficient to determine that a desired property has been violated, or even the system state in which it is violated. It is common for the sequence of events that leads to that state to be...
long and convoluted. It is important that the technique used to identify the property violation also be able to identify a specific sequence of events leading to the problem.

Tools exist to perform this detailed level of formal verification using various underlying techniques such as state space exploration and theorem proving. However, these tools are only as good as their ability to analyze realistically sized system. Real-world applications usually prove quite challenging for the tools. Sophisticated techniques are needed in order for the tools to be useful, and only a small number of tools are able to meet the needs.

**Use automation where possible to minimize human errors**

It is inevitable that humans make mistakes, so the best strategy for minimizing these mistakes is to minimize the places in the development process where humans have direct influence. In other words, automating the parts of the development process that are amenable to automation can significantly increase quality and security.

 Automation helps most in replacing or aiding humans for repetitive tasks, consistent application of rules and guidelines, keeping track of many details at once, and choosing among alternatives based on complex quantitative information. In software development, there are many such areas. For example, tools exist to determine if requirements are missing or inconsistent, create comprehensive tests suites directly from requirements, combine features into a final product based on the feature interactions, apply cross-cutting concerns such as authentication and logging, perform design analysis, aid in the analysis of legacy code, generate optimized code directly from design documents, target code to a given platform and framework, create protocol stacks from their specifications, and examine code for common weaknesses.

 While no automated tool is perfect, this automation has two main advantages related to software assurance. First, common situations that lead to vulnerabilities, such as off-by-one errors that cause buffer overflows, can be largely eliminated. Second, even though the automation tools themselves will inevitably have defects because they were created by humans, once a defect in a tool is fixed, it is fixed forever. This has the added benefit of then being able to re-apply the tool to fix all of the defects induced by that tool’s defect.

**Reduce the complexity of the design by using feature-based design**

One of the main enemies of software assurance is complexity. While a large number of cybersecurity issues relates to simple coding issues (e.g., failure to consistently check a return value), a significant portion of the issues can be directly attributed to the complexity of modern software systems. This complexity causes issues by pushing the number of things a human can keep track of at once, by introducing subtle and unintended interactions between parts of a system, and by obscuring abuse cases and timing issues (e.g., time of check to time of use (TOCTOU) errors).

 There are two main (complementary) strategies for managing system complexity: minimize the number of things a person must keep track of at once, and make each thing less complex. In order to minimize the number of things a person must keep track of at once, a system can be designed as a composition of individual features and their interactions. In this way, each feature can be specified, tested, and debugged separately. The interactions between the features can then be explicitly stated. Automated tools can then be used to correctly combine the features into a complete system. The second strategy is discussed in the next section.
Reduce the complexity of the design by hiding unnecessary details in the design

The design of a system can be made less complex by allowing the designer to focus on the important details while allowing the unimportant details to be left to automated tools. That is, although a certain amount of complexity is inherent in a system, one should minimize the incidental complexity that compounds the problem.

This focus on the appropriate level of abstraction is analogous to the popularity of higher-level languages, albeit that is usually couched in the framework of productivity instead of security. No one today would seriously contemplate building a modern software system in microcode or even assembly. The abstractions of the higher-level languages serve three purposes: they allow the developer to focus more directly on what the task at hand is instead of handling the details of how it is done, they allow for the encoding of best-practice implementation as embodied in the compiler/interpreter, and, as with other automation technologies, they remove a class of errors.

When the level of abstraction in the design closely matches the system domain, the incidental complexity of the design can be minimized. Many domain-specific languages (DSLs) exist, covering a broad range of domains. Examples include tabular notations for protocols, point-and-click interfaces for firewall configuration, graphical notations for combining mathematical functions, and state machine notations for state-based systems. These notations, combined with the appropriate tool support, can lead to significantly improved software assurance.

Reduce coding errors by rigorously following best-practice coding rules

It is telling that multiple lists of common code weaknesses exist, no less that there are myriad programs, both open source and commercial, available to check for them. Put the other way, failure to follow best-practice coding rules is still a major problem. Arguably, very little progress has been made in this area over the past several decades, and checking code after it has been developed is treating a symptom instead of the cause.

Whether the coding rules are not followed due to lack of awareness, laziness or forgetfulness on the part of the developer, misunderstanding of the rules, not understanding how to apply the rules consistently, or simple human mistakes, the result is the same. Given the obvious failure of trying to handle this problem at the human level, automation must be used to combat this problem.

A rule-based code generation system can apply best-practice and organization-specific coding rules universally and consistently during the creation of source code from a specification. Examples of coding rules that can be automated (i.e., common weaknesses that can be avoided) include avoiding buffer overruns, proper recovery from memory exhaustion and other exceptions, array bounds checking, protection from integer overflow and underflow, and the universal checking of return values.

Create and apply a comprehensive test suite guaranteed to cover all the requirements

There are many definitions of what coverage means for a test suite, including how well the tests exercise conditions that should never happen. However, no matter what the exact metric, test suites are only effective to the extent that they test all of the requirements. The best way to ensure requirements coverage is to derive the test scenarios from the requirements themselves.

It is possible to automatically derive a comprehensive test suite from the requirements when they are in UCM notation (described above). Analysis tools can guarantee that all branches of the UCM have been traversed at least once and therefore that all responsibilities have been exercised. Perhaps equally as important given the potentially infinite number of such test scenarios that can be derived, tools can
minimize the number of test scenarios required to provide this coverage. Using this derived set of test scenarios as a base, one can effectively and efficiently create a test suite for the final code that satisfies the required coverage metrics (e.g., branch coverage or symbol coverage).

**Ensure transparency of design and code such that malicious code cannot be inserted**

As a corollary to the need to reduce incidental complexity, a simpler design makes it more difficult to hide malicious features in the system. For example, a protocol stack based on the protocol specification itself would make it much more difficult to insert a covert back door based on secret handling of a specially formatted message.

While one cannot make it impossible for malicious code to be inserted, to do so would require that the malicious functionality escape detection in the design notation. A domain-specific notation coupled with a feature-based design and rule-based code generation present a high barrier for entry.

**Reengineer the legacy system to add modern software assurance practices.**

Typically, the older an application is, the less likely it was designed with cybersecurity concerns in mind. Depending on several factors, three main options for analyzing and addressing the cybersecurity of legacy code are available. First, if no source code for the application is available, traditional analysis techniques will not apply. In this case, use of a cybersecurity wrapper may be the only viable option (see below). Second, if the full source code for the application is available, a static code analyzer for the language and platform is available, and the static analysis tool covers the specific security concerns about the application, then third-party tools static analysis tools can be used. Note, however, that the commercial static analysis tools tend to look only for common weaknesses in code and not for higher-level concerns such as the possibility of confidential data leaking to an insecure channel. In addition, fixing the uncovered weaknesses will involve modification of the existing code, which in itself leads to numerous problems. Third, in all other cases (which may account for the majority of the legacy code under consideration), specialized analysis of the source code must be done.

Specialized source code analysis can take multiple forms, but it has two common characteristics: it must be driven by a subject matter expert (SME), and it is intractable and/or economically infeasible without support from tool automation. Automation helps maximize the effectiveness of the SMEs to increase the return on investment for the effort. Beyond the cost savings, automation is also critical the help the SME understand the complex interactions in any large application – a task for which humans have very limited capacity.

Semi-automated and proven capabilities for modernizing legacy code exist. This process generally consists of two major phases: reverse engineering to extract relevant information from the legacy code, and reengineering using the extracted information and quantitative analysis to refactor the code to improve various metrics (for example, coupling and cohesion).

This approach has direct advantages in analyzing legacy code for cybersecurity issues and addressing the issues found. It can be applied to any computer language, even proprietary ones, so it can be used on applications for which no static analysis tool exists. The application also does not need to be fully complete and compilable, as is the requirement with several commercial tools (although the code needs to be “complete enough” – no system can analyze code that does not exist). Cybersecurity concerns in themselves are features that can be analyzed and extracted by tools, and the concerns can be refactored and universally applied as part of the reengineering phase.
Ensure that the effects of compromised code are kept as local as possible

On the assumption that a system may be compromised, it is important to minimize the effects of any successful attack. An especially vexing problem is the use of a zero-day exploit against a large number of deployments of a given system. In effect, widely deployed software is a high-value target, and a single uncovered vulnerability can have widespread consequences.

One strategy for mitigating these kinds of exploits is to minimize the possibility that the same attack can be used on other systems, or even a second time on the same system. For the class of vulnerabilities based not on the logic of the code (e.g., forgetting to authenticate a user) but rather on the structure of the code (e.g., stack smashing), it is possible to make related exploits be usable only on a single deployment.

Using automatic code generation, each instance of the code can be unique while keeping the functionality identical. With unique code instances, even if one instance of the code is compromised through a technique such as stack smashing, the particular exploit is only usable on that one instance. There is an added benefit of unique instances, too, in that if an exploit is detected being tried, one can determine which specific instance of the code was originally compromised.

Since automatic code generation allows the application to be maintained at the design level instead of at the source level, the increased maintenance burden of multiple code instances is minimal. The code generator can make structural changes such as function inlining, loop unrolling, replacement of tail recursion, restructuring of code blocks, etc., in a manner transparent to the developer.

Provide an application-aware security wrapper during the execution of the legacy code

Through a combination of application-specific information and tight coupling with a virtual machine or operating system, a robust and systematic solution to legacy security concerns can be created. An insecure application can be protected from cybersecurity threats through a customized framework without having to modify and recompile the original application code. Many of the vulnerabilities of legacy applications are ultimately related to potentially incorrect or insecure uses of external interface points within the application. These interface points may be protocol stacks, system calls, library calls, user inputs, etc. However, only a small portion of these vulnerabilities are due to the use of any given interface point. Rather, the vulnerabilities are generally due to how the interface point is used. For example, the use of `fopen()` is not in itself a problem, but using it to open a file based on an unchecked, user-supplied (tainted) file name is a vulnerability.

A viable solution integrates a custom-created cybersecurity wrapper for the application that, in conjunction with tight virtual machine or operating system integration, sanitizes the uses of the interface points. Code generators can use supplied information about the application to create code that performs sophisticated, semantics-based interface checking. This interface checking can mediate many common vulnerabilities, including errors in the handling of protocols and other inputs, use of insecure protocols or functions, lack of appropriate authentication, etc.

UniqueSoft Software Assurance

UniqueSoft LLC, a US-based company with deep experience in developing software for high-availability and public-safety communication systems, has developed technologies and processes based on the lessons learned from more than two decades of system deployment. The UniqueSoft technologies fully leverage automation and cover new development as well as sustainment and modernization efforts.
The table below summarizes UniqueSoft’s offerings related to software assurance as discussed in this white paper.

<table>
<thead>
<tr>
<th>Potential Solution</th>
<th>UniqueSoft Offerings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide a high-level and easy-to-review notation for capturing the complete use cases of the system.</td>
<td>Capture and analysis of requirements using tools based on the ITU-T Z.151 Use Case Map (UCM) standard notation.</td>
</tr>
<tr>
<td>Perform formal requirements verification.</td>
<td>Formal Verification and Validation proofs of consistency, completeness, safety properties, and liveness properties of requirements.</td>
</tr>
<tr>
<td>Use automation where possible to minimize human errors.</td>
<td>Software services based on a proven automation tool suite. Automation tools cover requirements capture and analysis, test suite generation, model and code testing, feature composition, simulation, design analysis, code generation, and legacy code analysis and modernization.</td>
</tr>
<tr>
<td>Reduce the complexity of the design by using feature-based design.</td>
<td>Integrated tool suite for capturing feature-based designs, testing the features independently, and composing them into complete application designs.</td>
</tr>
<tr>
<td>Reduce the complexity of the design by hiding unnecessary details in the design.</td>
<td>Integrated tool suite for capturing designs in ITU-T Z.109 notation (a UML profile) and generating optimized and secure code in C, C++, C#, and Java targeted to a specific platform.</td>
</tr>
<tr>
<td>Reduce coding errors by rigorously following best-practice coding rules.</td>
<td>Rule-based code generator that applies not only industry best practices, but also is highly customizable so that new rules can be easily added.</td>
</tr>
<tr>
<td>Create and apply a comprehensive test suite guaranteed to cover all the requirements.</td>
<td>Creation of test scenarios from UCMs that, with a minimum number of tests, will cover all functional requirements.</td>
</tr>
<tr>
<td>Ensure transparency of design and code such that malicious code cannot be inserted.</td>
<td>Designs based on the ITU-T Z.109 UML profile, and rule-based code generation.</td>
</tr>
<tr>
<td>Ensure that the effects of compromised code are kept as local as possible.</td>
<td>Generation of multiple variants of an application that all have the same functionality but differ structurally.</td>
</tr>
<tr>
<td>Re-engineer the legacy system to add modern software assurance practices.</td>
<td>Complete legacy re-engineering services.</td>
</tr>
<tr>
<td>Provide an application-aware security wrapper during the execution of the legacy code.</td>
<td>Creation of security wrappers based on the characteristics of the original legacy code.</td>
</tr>
</tbody>
</table>

1 http://cwe.mitre.org