SYM-AM-18-091

PROCEEDINGS
OF THE
FIFTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

THURSDAY SESSIONS
VOLUME II

Acquisition Research:
Creating Synergy for Informed Change

May 9–10, 2018

March 30, 2018

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Automated Methods for Cyber Test and Evaluation[1]

Valdis Berzins—is a professor of computer science at the Naval Postgraduate School. His research interests include software engineering, software architecture, reliability, computer-aided design, and software evolution. His work includes software testing, reuse, automatic software generation, architecture, requirements, prototyping, re-engineering, specification languages, and engineering databases. Berzins received BS, MS, EE, and PhD degrees from MIT and has been on the faculty at the University of Texas and the University of Minnesota. He has developed several specification languages, software tools for computer-aided software design, and fundamental theory of software merging. [berzins@nps.edu]

Abstract

Cyber security of mission-critical software is a relatively new concern that is difficult to measure and hence difficult to incorporate effectively in software development contracts. The DoD has typically relied on black-box approaches to software testing. However, cyber vulnerabilities, particularly those deliberately injected into systems, are often statistically invisible with respect to affordable levels of black-box testing, which implies that they cannot be effectively detected using conventional testing techniques. This motivates augmenting traditional testing approaches with additional types of test and analysis procedures. This paper explores application of automated testing and other automated analysis methods to reduce cyber risks. We analyze several types of undesirable software behaviors and identify automated methods that could detect them within practical limits on time and computational resources.

Overview: Cyber Testing Challenges

Failures Are Not Random

The quality objectives for cyber concerns are seemingly similar to those for software meant to operate in uncontested environments, but on closer examination, there is a fundamental difference with far-reaching consequences for testing and evaluation. In both cases, we wish to minimize the risk of improper software behavior, consistent with the policy set in ICD 503 (Office of the Director of National Intelligence, 2008). However, “risk” has very different meanings in the two contexts.

For uncontested environments, failures act like random processes, and an appropriate risk concept is a statistical combination of severity of consequences from each type of potential mishap weighted by their frequency of occurrence, consistent with the formulation in DoD (2012). The ideas of “safety” and “reliability” are based on this point of view, which equates risk to the expected loss, damage, or injury when averaged over time. The unstated assumption of most work in this camp is that the probability distribution for mishaps is stationary, which means that the frequency of occurrence is stable over long periods. This approach is reasonably consistent with the properties of failures that are due to unintentional events with unpredictable variations, such as equipment wearing out, human errors, electronic noise, background radiation, and so on.

[1] The views presented in this paper are those of the author and do not necessarily represent the views of DoD or its components.

In contested environments, failures are not random, and frequency of occurrence can vary dramatically depending on external circumstances, driven by the existence of an adversary whose deliberate behavior depends on variable conditions, such as the following:

- Are we at war?
- How much profit/military advantage/political value would a successful attack provide?
- Are sufficient resources available for successful attack?
- How much risk of prosecution or counterattack is there?

In this case, which matches cyber-attacks, game theory provides a better model of risk than statistics. The associated underlying assumption is that there is a capable adversary who will choose courses of action that maximize damage. Consequences of this paradigm shift include the following:

- Focus of risk management shifts from minimizing expected loss to minimizing worst-case loss.
- Scope of risk management expands from mitigations concerned solely with the software to those that address both the software and the adversary.
- Risk assessment becomes sensitive to surprises due to new adversary tactics.
- Unlikely conditions and rarely traversed paths through the code can no longer be ignored.

Causes and Effects Will Be Hidden

A consequence of Rice's theorem is that perfect cyber certification is impossible. This theorem is a well-known result in computability theory that says any non-constant property of program behavior is undecidable. Non-constant means the property is true for at least one program and false for at least one other program, which holds for all software security properties of interest. "Undecidable" means there is no systematic method (algorithm) that will always produce a correct decision and will always terminate in a finite amount of time. This theorem applies to both testing and static analysis of the source code.

Since any workable certification procedure must fit into a definite schedule, it must operate within some reasonably short bounded time, say less than a year. The theorem therefore implies that all practical certification procedures produce imperfect decisions: Any such procedure must produce some false positives and some false negatives, or fail to reach a conclusion for some inputs.

Capable adversaries expect defenders to search for vulnerabilities, know that detection will be imperfect, and design their attacks to make them difficult to find. Some ways to hide are as follows:

- Small footprint: Design malicious behavior so that it will be triggered in only one of a huge number of possible execution conditions. The triggering condition can be made statistically invisible very easily because the search space has exponential size[2] (see examples in Berzins et al., 2015).
- Fragmentation: Malicious behavior resulting from interaction between widely separated parts of the code, such as exception handlers and multiple threads with no logical connection. Such non-local interactions are difficult to detect in large systems.
- Delayed manifestation: Corrupt the code or data in ways that will not affect behavior until much later, possibly waiting for a statistically invisible external trigger.
- Timing: Information content of behavior is correct, but is delayed sufficiently to cause failures. Software that controls physical components is susceptible to this hazard.
- Parasitic effects: Breaking the model of computation so that logically correct source code can produce damaging behavior.

Consequences Are Physical

Much work on cyber security focuses on information—how to keep it confidential, free from corruption, authentic, and so forth. However, risk-based approaches are guided by severity of consequences, and relevant consequences are physical. Thus, system context must be considered as part of risk management, and if context is expanded far enough, all critical software is part of some cyber-physical system and either controls or influences its behavior. The following are examples of possible consequences of software faults:

- Dangerous physical events involving controlled equipment, such as collisions between moving vehicles or discharge of weapons
- Disrupting defensive capabilities of a military platform in a conflict, increasing vulnerability to kinetic attack
- Revealing the location of a military unit or identity of a covert operative, exposing them to attack
- Revealing military plans, enabling adversaries to target weak points and increase damage
- Economic and political analogs of the above

[2] 2^b cases to test, where *b* is the number of bits in all input and state variables combined.

Threats Can Morph

The most serious cyber risks are due to corruption of software at runtime. Runtime corruption of the executable code can result in unlimited damage at the discretion of the adversary. Not only is severity of consequence potentially the worst possible, but also, detection of this type of threat is especially challenging because the potentially damaging behavior is inserted after certification processes are completed. The destructive payload is not present either in the source or in the executable that is analyzed, so there are no adverse consequences to detect in the initial uncompromised configuration of the system under test.

Recovering From Mishaps

The above lines of reasoning imply that software defenses will never be perfect, especially for systems of practical size, for which exhaustive analysis is impractically expensive both in terms of money and in terms of available time (billions of years may not be enough).

Strategies involving defense in depth can be useful for extending the time that systems under cyber-attack can remain operational, but they cannot provide complete immunity to attack by a capable adversary. This well-known approach to cyber-defense needs to be augmented with self-healing capabilities—so that systems can recover from partially damaged states and continue operating despite partially successful cyber-attacks. Even better are adaptive immune responses that reconfigure a system damaged by attack so that it is no longer vulnerable to a replay of the previously successful attack. The objective should be to increase the time and cost of successful attacks to the point where they are not affordable by adversaries.

Some self-healing capabilities are practical at the current state of the art, and continued research to improve this approach is recommended.

You Don't Know What You Don't Know

Adversaries are constantly finding new ways to attack systems. This makes it difficult to write development contracts for secure software, because contracts inherently define fixed responsibilities for the contractors, but the set of actual threats is incompletely known and open-ended. Supporting a rapid repair capability requires agility both in the software and in the contracting approach.

This seemingly unsolvable problem can be addressed by suitable application of Open Systems Architecture and Technical Reference Frameworks. Rapid reconfiguration can be supported by an architecture that accommodates all needed configurations without changing the architecture. In this context, architecture can be considered to consist of the aspects of a dynamic system *that do not change*. Architecture for cyber-resilient systems should include standardized structures and requirements for supporting functions related to resilience, such as runtime monitoring and self-healing functions.

As an example, consider the high-risk threat of runtime code compromise. Services called out in associated parts of a Technical Reference Framework should include facilities for the following:

- Secure, authenticated distribution of software updates
- Runtime monitoring of executable code to detect unauthorized changes
- Restoring corrupted code to an authorized configuration
- Restoring the execution state to a valid configuration and resuming execution with the restored code configuration

These services and modules should all conform to a stable standardized interface specified in the architecture, so that best-in-class components providing these services can be shared across different systems and future technology improvements related to these critical issues can be readily incorporated by software module swapping, ideally without stopping operation of the system. The example illustrates a vision of how rapid reconfiguration capabilities could be specified in a fixed architecture fragment that could be called out in fixed and definite development contracts.

Insider Threats

Turn-Key Malware

Malicious insiders may build some types of malware into software before delivery. These people are part of the development team and have full access rights to the code. Examples of this type of malware include "Easter Eggs," which are extraneous bits of functionality that are typically triggered by some single special input value, often one that is extremely unlikely to be encountered as part of a normal workload. Although many known instances of Easter Eggs have done little harm, the ability to detect the pattern is a cyber-testing concern because Easter Eggs can also hide extremely damaging extra capabilities, such as enabling unauthorized access or unauthorized modifications to a system.

Testing Difficulties

Common testing practices in the DoD rely heavily on black-box testing, in which test cases are designed based on the requirements, without knowledge of the structure of content of the source code. The method is widely used because it is reasonably good for checking that the delivered software has the behavior specified in the requirements, which is a primary concern in acceptance testing. It may also be the only viable testing approach if the development contract does not include rights to access the source code for the developed software.

Unfortunately, black-box testing is not effective for checking the absence of undesired extra functionality, such as deliberately planted cyber vulnerabilities. Since there is often only a single test case that could demonstrate the existence of an embedded Easter Egg and the number of possible test cases is usually astronomical, the odds of detection by black-box testing are practically none (see Berzins et al., 2015, for quanititative details).

Solutions

Clear-box testing with respect to the statement coverage criterion can detect Easter Eggs in a practical manner that can be readily specified as a development requirement in a contract. The statement coverage criterion requires that every statement in the source code must have been executed by at least one test case. Relatively low-tech tools that count the number of times each statement in the code has been executed can check compliance with this requirement; the requirement is met if all of these numbers are greater than zero, which can also be checked by a simple piece of software. Many compilers include options for measuring statement coverage, for example, the gcov facility in the gcc tool set, which directly reports the percentage of source statements that have been covered by a test ("Monitoring Statement Coverage," n.d.).

Difficulties with this approach include finding test cases that can exercise rare paths and handling unreachable sections of the code. Although the general problem of finding test cases that trigger particular paths in the code does not have an effectively computable solution, experience with fuzz testing shows that constraint solver tools can handle a majority of the cases that arise in practice (Cadar et al., 2008). Additional tactics that can be useful for exercising rare paths are to seek module-level test cases that reach the

statements in question, as opposed to system-level test cases. This provides more direct control over the execution state of the module and simplifies the constraints that need to be solved to generate the needed test cases, enabling a larger fraction of the cases to be solved automatically.

In some cases, unreachable code as well as extraneous code that does not affect the outputs of software services can be identified by using a form of dependency analysis known as software slicing (Berzins, 2014). These automated methods can help diagnose parts of the code that could not be exercised by test cases. The remaining cases are a small fraction of the code and may be few enough to be affordably examined by human analysts.

Outsider Threats

Runtime Code Modification

As noted above, runtime code modification is the most serious cyber risk in any system, because its severity of consequences includes the consequences of all other cyber risks. From a game-theoretic viewpoint, which focuses on worst-case risk, we expect an adversary to inject the most damaging exploit available to them if they chose a runtime code modification attack.

The signature of a runtime code modification vulnerability is not presence of inappropriate software behavior, but rather the existence of a possible path for executable code (or data that affects code behavior) to be eventually modified without authorization. The triggering condition as well as the inappropriate actions in such a path may involve interactions that bypass the official interfaces of the system to be certified, and may not be present in the high-level models programmers use to design and check their code. For example, the triggering event may involve corruption of system memory from a logically unrelated function or process. This implies that this type of malicious behavior may be completely invisible to black-box testing in the initial uncorrupted state of the software, not just statistically invisible.

Another consequence of this cyber risk is that the critical parts of the code cannot be localized within current development approaches, especially in the context of programming languages without garbage collection and provisions for memory protection. Any part of the software that manipulates pointers can be a potential avenue for attacks that modify code at runtime, and these parts are spread throughout most systems. This makes it very difficult to focus the most intensive testing and analysis efforts on just the “security critical” parts of the system, and greatly increases testing cost for high confidence systems.

Detection

Following the principle of defense in depth, we suggest a layered approach to detection, coupled with mitigations that combine preventive and remedial measures. This section focuses on detection. The following are possible measures:

- Software update service analysis
- Architecture conformance checking
- Memory allocation checking
- Memory reference checking
- Runtime monitoring of executable code

Many current systems include explicit interfaces for upgrading the software to new versions or distributing patches. This part of the system is cyber-critical, expected to be a

focal point for attack, and should be subjected to the most intense degree of testing and analysis possible, at multiple levels.

- At the requirements and architecture level, check whether the service is required to authenticate authority to update the software and check integrity of the transmitted code, and whether the methods for doing so are the strongest available at the time. The latter review has to be repeated regularly to account for future development of improved methods for providing these capabilities.
- At the source code level, do static analysis of the implementations of these methods, up to and including constructing/checking mathematical proofs of the security properties of the protocols and algorithms used for transmitting and installing the software updates. Also check that the source code matches the algorithms and protocols that were proved, if the proofs are done based on some representation other than the actual source code for the service.
- At the executable code level, do penetration testing by highly competent red teams aimed at these services, and check that the executable code matches the source code and is free from extra functions. This last step is needed to guard against possible compromise of the compilers, linkers, and loaders used to build the software, which could be corrupted to add back doors into the executable code they produce, which could be specifically targeted at just the implementation of the software update service. Such back doors could bypass the authentication and integrity checks that exist in the source code and provide unauthorized access to the software update service when activated by adversaries.

Checking conformance to architecture includes checking that interfaces and executables do not contain any extra services or interaction paths, beyond those specified in the system interfaces. Such extra services could be avenues for execution of malware, and extra interactions could be paths for triggering malware or exfiltrating its results. Architecture conformance has two levels: checking actual source code interfaces for extra services, and checking source code of services that are specified in the architecture for extra interactions, such as reading or writing from files, network locations or global variables that are not included in the interface specification for the service. This latter check can be done via dependency analysis algorithms such as data flow analysis and software slicing (Berzins & Dailey, 2009).

Memory allocation checking consists of checks for references to pointers that have not yet been initialized or that point to memory areas that have already been deallocated (“dangling pointers”). This is a common problem in programming languages without automatic garbage collection, and there exist commercial tools for doing such checking, including Insure++ and Valgrind (“Parasoft Insure++,” n.d.; “Valgrind’s Tool Suite,” n.d.).

Memory reference checking is a runtime check that all pointer references refer to a non-null object of the proper type and that all array references are within the range of declared array bounds. Compilers of some languages can do this, and for some such as Ada, the runtime checks are required except for contexts in which the compiler can prove they are unnecessary because violations are impossible. Requiring use of such facilities would make it more difficult for adversaries to create exploits that corrupt memory containing executable code and critical data. Proof systems such as Spark can check properties such as these mostly automatically (“Spark Pro,” n.d.).

Runtime monitoring of executable code is an active check at runtime that the executable code matches the most recent version that was installed (Berzins, 2014). There should be a Technical Reference Framework that specifies a standard service for doing this, which can be easily incorporated in the architecture of any mission-critical system. This would enable development of standard software implementations of this service that could be used in any system conforming to the architecture. These implementations could have variants that work with different operating systems and different programming languages.

Mitigation

Detection of vulnerabilities and attacks is not sufficient for achieving reliable system operation—mitigation and recovery are needed as well. Some mitigations for code corruption attacks are as follows:

- Using pure code segments contained in read-only hardware. This preventive measure would make runtime code modification attacks impossible, at the expense of specialized hardware and prohibition of automatic installation of software updates. This approach is not a new idea—it was used in very early systems that had magnetic drums as secondary memory (predating magnetic disks). A modern version could use erasable programmable read-only memory (EPROMs), which could allow updates but only with human intervention via physically exposing the memory chips to UV light to erase them and enable them to be reprogrammed. This would make software updates less automatic, but could provide two-factor authentication for updates and guarantees of absence of change between such updates. This mitigation is relatively expensive and cumbersome, but it could provide very high levels of runtime code protection to critical applications that really need it.
- Restoration of code from ROM. A lighter-weight version of the above mitigation is to require runtime monitoring of the executable code, as described in the previous section, together with a facility for restoring the code from a backup copy in read-only memory (such as a locked CD ROM), restoring a safe execution state, and continuing the operation of the software. This solution can be implemented using existing hardware, but would require more time to restore operation after a failure. A stronger version of this idea would use a backup copy with a different code layout, which would reduce the chances that replay of the same attack would succeed again, thereby increasing time to next failure.
- Disabling reflective language capabilities. Some modern programming languages, including Java, provide capabilities for runtime inspection and modification of interfaces and implementations. Access to such capabilities makes an adversary's job much easier. Development contracts should either require the use of a programming language without reflective capabilities, or require the developer to demonstrate that those capabilities have been removed from the system.
- Use of programming languages with garbage collection should reduce exposure to the threat of code and data corruption. The memory allocation and recycling facilities of languages and their supporting systems (compilers, runtime libraries, linkers, loaders, etc.) should be intensively checked for faults that could lead to memory corruption in applications constructed using those languages and systems.

Conclusions

There is no silver bullet when it comes to cyber security, and no such thing as a completely secure system. The best practical solutions involve a layered set of defenses and mitigations that increase the time and effort it will take an adversary to compromise the system and decrease the time to detect a compromise and restore dependable operation. An appropriate goal would be to make system compromise prohibitively expensive for most, if not all, potential attackers.

This paper defines a risk concept appropriate for gauging cyber threats, identifies cyber risks with greatest risk exposure, and suggests corresponding methods for detection and mitigation. In addition to methods that make the systems more difficult to compromise, we recommend further investigation of mitigations that address both the system and the adversary. This would include stronger methods for authenticating access to systems and networks, along with facilities for recording and linking people's identities to evidence of potential wrongdoing that could support deterrence in the forms of legal prosecution of individual wrongdoers and determined public action against state-sponsored attacks.

References

Berzins, V. (2014). Combining risk analysis and slicing for test reduction in open architecture. In *Proceedings of the 11th Annual Acquisition Research Symposium* (NPS-AM-14-C11P07R03-038; pp. 199–210.). Monterey, CA: Naval Postgraduate School, Acquisition Research Program.

Berzins, V., & Dailey, P. (2009.) How to check if it is safe not to retest a component. In *Proceedings of the Sixth Annual Research Symposium—Acquisition Research: Defense Acquisition in Transition* (pp. 189–200). Monterey, CA: Naval Postgraduate School, Acquisition Research Program.

Berzins, V., Van Benthem, P., Johnson, C., & Womble, B. (2015). Use of automated testing to facilitate affordable design of military systems. In *Proceedings of the 12th Annual Acquisition Research Symposium*. Monterey, CA: Naval Postgraduate School, Acquisition Research Program.

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth USENIX Conference on Operating Systems Design and Implementation* (pp. 209–224). Retrieved from http://static.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar_html/paper.html

DoD. (2012). *Standard practice system safety* (MIL-STD-882E). Retrieved from http://www.system-safety.org/Documents/MIL-STD-882E.pdf

Monitoring statement coverage with gcov. (n.d.). Retrieved from https://www.cs.odu.edu/~zeil/cs333/website-s12/Lectures/wbtesting/pages/gcov.html

Office of the Director of National Intelligence. (2008). *Information technology systems security risk management, certification, and accreditation* (Intelligence Community Directive Number 503). Retrieved from https://www.dni.gov/files/documents/ICD/ICD_503.pdf

Parasoft Insure++. (n.d.). Retrieved from http://www.parasoft.com/jsp/products/home.jsp?product=Insure

SPARK Pro. (2012). Retrieved from https://www.adacore.com/sparkpro

Valgrind's tool suite. (n.d.). Retrieved from http://valgrind.org/info/tools.html

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net