Application Security: Protecting the Soft Chewy Center

Alec Main
Cloakware

The software at the heart of many military systems is traditionally defended using the network security model in which vulnerable processing and information are encircled in defensive technologies. Unfortunately, network security is proving insufficient to defend against a wide range of attack scenarios targeting commonly available software. Likewise, system-level strategies are proving just as insufficient to defend against such attacks. However, effective application security provides protection from the inside out by implementing defensive techniques into applications and data themselves. As more commercial off-the-shelf hardware and software is used for military purposes, application security becomes more important since the skills and vulnerabilities that can be leveraged in attacks are more widely known. This article applies commercial lessons learned to military scenarios.

Software is at the core of all our military systems that provide technological advantage and strategic superiority over adversaries. Not only is software essential in delivering technological advantage, it also represents a significant portion of the defense program's operational capacity – and a significant investment.

When software applications fall into the hands of adversaries, they can analyze weapons, tactics, techniques, and procedures for vulnerabilities, and can produce countermeasures and more lethal weapons while saving research and development costs [1]. Military forces must prevent their information and weapons assets from being turned against them.

In the past, systems built with proprietary hardware and custom software were inherently more difficult to attack or reverse engineer. This approach, however, has become less desirable as commercial off-the-shelf (COTS) hardware, operating systems, and applications provide richer features, performance, and flexibility while reducing costs and deployment time.

Modern COTS software is complex, with a Windows or Unix operating system and major applications consisting of over 100 million lines of code. At an estimated frequency of one security bug per thousand lines of source code, a typical system will have over 100,000 security vulnerabilities [2]. By their nature, COTS systems are also widely available to and well understood by hackers. Information about vulnerabilities is widely shared, and tools for reverse engineering are readily available on the Internet. Military adoption of COTS software makes it potentially easier for foreign adversaries to reverse engineer and steal the technological advantage embodied in these software applications.

These facts are recognized by the Department of Defense (DoD). In December 2001, the Software Protection Initiative (SPI) was established to prevent the exploitation of national security application software by U.S. adversaries. As a U.S.-led initiative, the SPI is on the leading edge of determining the requirements for application security and guiding development of protection techniques. In addition to the two traditional components of information assurance – network security and operating system integrity – the SPI recognizes that an application-centric approach to protecting important DoD software is an essential *third leg* to the information assurance triad [3].

Inherent application security is also vital for critical infrastructure protection at home. Energy and utilities, communications, financial networks, transportation, and emergency and government services all depend on maintaining our applications and networks, which depend on software that currently lacks this third element in the triad of information assurance.

The DoD is acting to protect its software technology from reverse-engineering, unauthorized use, theft, and other types of exploitation. This article addresses modern techniques to make software applications inherently secure, and explores ways to protect vulnerable applications and the data they handle, including real-world examples.

The Need for Software Protection

Commercial and military network security specialists are beginning to recognize that application protection is a vital part of their overall security strategy. Perimeter security, in which we attempt to protect vulnerable assets in a trusted environment behind a secure perimeter, no longer works.

Firewalls were the first form of information technology perimeter defense, but they remain vulnerable to viruses and worms that penetrate and compromise internal systems using authorized network access vectors. Additional defenses such as virus scanners and intrusion detection systems are reactive and remain vulnerable to new threats. As networks expand, just defining the perimeter is challenging. Furthermore, perimeter defenses cannot protect application software from exploitation by insiders. With perimeter defenses breaking down, the result is software being deployed in inherently hostile environments.

Cryptographic technologies have long been used to protect data in transit and in storage. However, cryptography assumes that the end points, or points of use, are trusted. This is no longer the case as perimeter defenses such as firewalls break down and insiders can no longer be trusted. Compromised end-point systems can render even the best cryptography useless. Vendors of applications that present intellectual property in the form of music, movies, and games are representative of the realization that valuable encrypted data can only be fully protected by end-to-end security that encompasses the application software in addition to data encryption.

Application vulnerabilities are important for the military because they exist in one form or another in many military scenarios. History tells us that insider threats can never be ignored when designing military systems. COTS-based software compounds the threat by providing hostile parties with well-understood targets. But further, military software systems deployed in the field risk physical capture by adversaries.

Software protection [4, 5] is vital to defending these assets.

How Do We Protect the End-Points?

Currently, the common approach to protecting the end-points is to use system-level security by, for example, signing

and/or encrypting all the software and data with a cryptographically strong key. But this approach looks remarkably like the old approach to network security – a perimeter or fortress defense – with a soft, chewy, vulnerable center.

The goal of system-level security is to build a *chain of trust* with the root of trust in hardware [6]. In this approach, the hardware validates the boot loader, which validates the kernel, which in turn validates the applications. Cryptographic [7] techniques are used to sign the code involved throughout the trust chain.

Code signing has its limitations. It is commonly used on the Internet to warn users of malicious software, called malware, and/or Trojan Horses that they might download. To be effective, the host must be trusted, every application must be signed, and the user must not want or – preferably – not even have the option to download unsigned software. However, the checking mechanism is typically not secure itself, meaning that code signing offers no protection when the host itself is untrusted or under attack.

Chain of Trust or House of Cards?

System security can be effective to a point; however, once broken, everything above the break, including the intellectual property and critical processing within the software applications, is exposed in much the same way as network security is breached once the firewall is penetrated.

System security is expensive to design and deploy in the first place and, once broken, is expensive and/or impractical to effectively and securely replace. Legacy issues also complicate upgrades needed to keep ahead of the latest hacking techniques. For these reasons, software-based renewable systems are being deployed by telephone companies for new, always-connected applications such as Internet Protocol Television (IPTV), where upgrades can be designed in, allowing controlled security updates to be pushed to the system across the network. Similarly, next-generation cable television security is moving to renewable software. Software-renewable systems through methods such as proactive obfuscation [8] are one means by which the military can deploy security that is both robust and affordable.

Example: Xbox

Microsoft's Xbox game console is an interesting public example of an intense reverse engineering and hacking effort [9]. Microsoft had two goals in locking down the Xbox: to prevent illegal copying of their games, and to prevent subsidy fraud by which a user could use the heavily subsidized hardware for unauthorized applications or purposes. The Xbox was designed to ensure that only signed applications could run, that only the original Microsoft approved code could be used on the Xbox, and that copied DVDs could not be used.

Microsoft realized that not only did the DVD authentication code need to be protected against reverse engineering, but the chain of trust needed to extend down to the boot loader to ensure that only a valid operating system could be loaded. Microsoft left the original boot loader in the ROM as a decoy that would entice attackers to waste their time. The true boot loader was then placed in the graphic controller and written to be self-verifying. The true boot loader contained the obfuscated key used to decrypt and load the kernel into memory. The kernel, in turn, verified that only signed applications were loaded and only original game DVDs were being used.

The huge popularity and ready availability of the Xbox and copy-protected games provided a large target for the hacker community. It took six months to finally break the protection. When the real boot loader was finally discovered and reverse engineered, the chain of trust came tumbling down. Copied DVDs and other applications could be run on the hardware. However, to replace the boot loader required a special *mod* chip that required cracking the Xbox cover and voiding the warranty.

Nevertheless, a second attack was then developed that exploited a buffer overflow, which meant no hardware modifications were necessary. Since code verification only occurred at boot-up, arbitrary code could be run independent of the system-level security.

The Xbox reminds us what happens when the chain of trust is broken and the house of cards comes tumbling down. The Xbox presented a challenging situation because the system was designed to be self-contained and autonomous – like many weapons systems or autonomous vehicles – and the user untrusted. Captured vehicles or undetonated bombs present a very similar type of threat, albeit infinitely more dangerous in the military arena.

But Will the Military Do It Differently?

Military systems can put more constraints on the user than a commercial producer like Microsoft can. One obvious improvement would be to include a hardware root of trust, either in a security chip such as a Trusted Platform Module, or a removable component like a smart card or dongle. The theory behind this approach is that the smart card can be removed or even physically destroyed by a trusted user prior to capture of the system.

The system must be designed to ensure all the appropriate software is encrypted and that a necessary secret lies in the removable key. Without the key stored on the smart card, the system cannot decrypt and run software applications. The system must still boot sufficiently to read the card, but if it is not present, an attacker will not be able to jeopardize the system.

But even this added level of protection has limitations that expose the military to potential security breaches. For example, the system could still be reverse engineered by a nation that legitimately purchased it and has access to the entire system. Without further precautions, the technology's soft, chewy center is exposed.

The same vulnerabilities would be exposed if the system were captured by an adversary when still running, or if the smart card were not removed, with serious consequences of disclosure and a possible weakening of other deployed systems.

A prime example is the U.S. Navy EP-3E Aries II surveillance plane that made an emergency landing on Hainan Island in China in April 2001 after a collision with a Chinese fighter jet. The plane was valued at U.S. $80 million and was packed with sophisticated eavesdropping equipment [10]. Military experts were concerned that Japan and the United States would have to change their secret communication system, at a cost of millions of dollars, as a result of Chinese scrutiny of the top-secret equipment [11]. Application security would have seriously hampered efforts to reverse engineer sensitive systems on board.

Missiles and bombs that do not explode are also vulnerable to reverse-engineering of weapons and guidance systems by adversaries. The United Nations mine clearing officials stated that between 10 percent and 30 percent of the missiles and bombs dropped on Afghanistan have not exploded [12]. According to a DoD briefing, a total of seven Tomahawk cruise missiles went astray in the Saudi Desert without exploding [13]. In Kosovo, Yugoslavia, evidence of missiles and bombs that had not exploded included a U.S. $1.25 million High Speed Anti Radar Missile on a highway and a Maverick anti-

tank missile that had apparently missed its desired target and embedded itself in the roadside verge [14].

Planning further defenses against these scenarios without application security exposes the difficulty of trying to patch together highly specific defenses against every eventuality. Would a special launch sequence be required for missiles, loading a decryption key into memory from a removable smart card before launch? How would field officers know if an autonomous vehicle were downed and how would they take action to defend the system? Would sensors track its state so that random access memory could be cleared and the system shut down? These approaches each bring operational scenarios and key management issues into a battlefield environment with associated training, readiness, and the need for correct execution. As keys proliferate, the insider threat only further increases.

Application Security Provides Defense in Depth

To be truly effective, there must not be a soft chewy center, but hardened applications that are an active part of the chain of trust. A chain that establishes real trust is a chain mesh rather than a series of links. The hardware authenticates the loader, which authenticates the kernel, which authenticates the application. The application in turn authenticates the hardware, loader, and kernel while it is running, preventing one break at a low level from providing the keys to the castle. Application security provides protection against insiders even if system-level security is compromised. Reaction when tampering or an attack is detected can be as varied as logging, *calling home*, or an intentional destructive hostile response such as erasing hard drives or damaging central processing units.

Application security further increases the effort required to break system-level security because failure is not directly correlated to attacker actions. For example, tampering detected in one part of the software may result in subtle data errors that only become obvious elsewhere, making it difficult for a hacker to know where the attack was discovered. Unlike the perimeter tactics borrowed from network security, this provides a logical and complex form of defense in depth.

There are a number of application-level security techniques available to defend both source code and compiled applications. Typically, the best strategy is to combine multiple defenses in ways that are complementary and protect not only the application and data, but also each other.

Application Security Techniques

To understand application security techniques, it is best to understand the following traditional means by which software is attacked.

- **Analysis.** Classic reverse engineering and analysis of the software and protocols to identify vulnerabilities. This can be static analysis when the code is not running such as disassembly and decompilation, or dynamic tracing of the executing code using debuggers and emulators.
- **Tampering.** Modifying the code and/or data so that it performs according to the attacker's objectives rather than as designed.
- **Automation.** The creation of scripts or code to apply the tampering attack to multiple copies of the application. These are also known as class attacks or global breaks.

Application security techniques are used to prevent both static and dynamic analysis, as well as static and dynamic tampering to the software (see Table 1). This technology includes the following:

- **Code obfuscation.** Data flow and control flow transformations to prevent reverse engineering and tampering.
- **White-box cryptography** [15]**.** Specialized obfuscation techniques for cryptographic algorithms that prevent secret keys from appearing in memory during cryptographic operations.
- **Integrity verification.** Robust on-disk and in-memory verification to detect static tampering of the entire executable and dynamic tampering of code.
- **Anti-debug.** Detection and prevention of the use of debuggers and emulators to prevent dynamic analysis.
- **Code encryption.** Just-in-time decryption of code in memory to prevent static reverse engineering and tampering. Decompilers are ineffective with these techniques.
- **Diversity** [16]**.** A random seed is typically used by the tools that implement the software protection technology such that a different random seed creates a structurally different result. Diversity prevents automated or global attacks from being developed by adversaries against the protection techniques.

The first goal is to make static analysis difficult, time-consuming, and/or expensive. The obvious approach to prevent static analysis is to encrypt the binary. While there are techniques to extract these decrypted executables from memory, there are also source code techniques that prevent static analysis such as control flow flattening, which introduces pointer aliasing that can only be resolved at runtime. In addition, there are specific decompilation and disassembly prevention techniques that target these tools. Note that while very powerful disassembly tools exist, most low-level code written in C or C++ is very difficult to decompile with only a few tools available. Software protection is about using multiple layers of defense and all these techniques should be considered.

Runtime analysis of a system can be made very time-consuming by using anti-debugger and anti-emulation techniques. A range of techniques unto themselves, these can be effective on targeted platforms. The code can be tied to the platform via node locking and loading of new applications controlled by secure code signing techniques. Advanced just-in-time decryption (or self-modifying code) techniques also raise the bar against dynamic analysis. Authentication of components on the machine and encryption of communication channels, with protocol not subject to replay attacks, also prevent analysis. In addition, data flow transformation techniques can be used to hide and randomize data values even when operated on within main memory. White-box cryptography refers to specific cryptographic implementations designed to prevent key extraction even when the operation can be statically or dynamically viewed by an attacker. Steganographic techniques can also be used for key hiding.

Static tampering is prevented with binary encryption techniques, as well as by

Table 1: *Application Security Techniques and the Types of Attack They Defend Against*

	Reverse Engineering Attacks		Tampering Attacks		Automated Attacks
	Static	Dynamic	Static	Dynamic	
Code Obfuscation	✓	✓	✓	✓	Diversity
White-Box Cryptography	✓	✓			Diversity
Integrity Verification			✓	✓	Diversity
Anti-Debug		✓			Diversity
Code Encryption	✓		✓		Diversity

introducing data dependencies in the code to change an easy branch jamming attack into tampering – increasing the effort required and involving multiple changes to the code. An important technique to prevent tampering is code signing, but the code signing mechanism itself will be subject to attack and so must also be suitably hardened. Integrity verification of applications should be done statically (on-disk) as well as in memory to prevent dynamic tampering attacks.

Prevention of automated attacks is best achieved by deploying code and data diversity so that a successful attack will only work for a subset of users. Diversity of code is a result of most software protection techniques outlined above. It is similar to having different keys (diversity of data) for different users. Diversity of code recognizes that attacks will be made on the software in addition to the data. Automated attacks are also mitigated by software renewability, which can be made low cost – if designed in upfront. Conversely, with hardware-based security, renewability is a major cost. Software can be renewed selectively, proactively, or reactively – depending on the strategy and the attacks to the specific system.

Diversity is a prerequisite for successful renewability; otherwise, attackers will perform differential analysis. This is a powerful attack used to quickly determine the changes made to software upgrades and shorten the time to successfully hack.

These application security techniques are integral to the SPI. The SPI's goals include institutionalizing software protection as part of the application software life cycle and developing user-friendly protection techniques. Institutionalized and easy-to-use software protection techniques like those described above provide an additional layer of security that helps to ensure the availability of critical assets and infrastructure while maintaining the strategic lead in technologies critical to national defense.

Institutionalized application-level security techniques can help assure that the development environment is safe against insider threats. For example, obfuscation of source code and diversification of binaries during the development process can help prevent a production system from being reverse engineered despite being protected, because earlier prototypes were not. Every version sent to the field for testing can be protected distinctly from every other version at low cost. This sort of technique can be particularly important for the development of complex systems that involve a significant software engineering factory. Innovations developed by subgroups within the process can be hidden from participants elsewhere in the product development cycle.

Conclusion

When developers build security directly into their applications, they significantly strengthen resistance to attacks, even on open platforms and in hostile environments. While system-level security can be effective to a point, it does not address all the attack scenarios, especially the insider attack. Rather than a chain of trust, application-level security combines with network security and operating system integrity to form a chain mesh. COTS systems lower system development and deployment costs while application security maintains the protection needed in today's networked, technically advanced warfare. Combined, they offer superior value and flexibility in the quest for battle readiness and operational success.

References

1. Hughes, Jeff, and Dr. Martin R. Stytz. "Advancing Software Security – The Software Protection Initiative." 8th Annual International Command and Control Research and Technology Symposium, Wright Patterson Air Force Base, Ohio, 2003 <www.dodccrp.org/events/2003/8th_ICCRTS/pdf/137.pdf>.
2. Trusted Computing Group. "Enabling the Industry to Make Computing More Secure." Backgrounder Jan. 2005 <www.trustedcomputinggroup.org/downloads/background_docs/TCG_Backgrounder_revised_012605.pdf>.
3. Trusted Computing Group.
4. Main, A., and P.C. van Oorschot. "Software Protection and Application Security: Understanding the Battleground." International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography. Heverlee, Belgium, June 2003 <www.scs.carleton.ca/%7Epaulv/papers/softprot8a.pdf>.
5. van Oorschot, P.C. Revisiting Software Protection. Proc. of 6th International Information Security Conference, 2003:1-13. Bristol, U.K., 2003. Springer-Verlag Lecture Notes in Computer Science, Oct. 2003: 2851.
6. Trusted Computing Group.
7. Menezes, A.J, P.C. van Oorschot, and S.A. Vanstone. Handbook of Applied Cryptography. 5th ed. CRC Press, 1996.
8. Schneider, F. B., L. Zhou. Distributed Trust: Supporting Fault Tolerance and Attack Tolerance. Jan. 2004.
9. Huang, Andrew. Hacking the Xbox: An Introduction to Reverse Engineering. No Starch Press, July 2003: 288.
10. Pan, Philip P. "U.S. Team Arrives in China to Examine Plane." The Washington Post 1 May 2001.
11. Chandler, Clay. "No Deal Reached On Plane: U.S., China Agree Only on More Talks." The Washington Post 20 Apr. 2001.
12. Ryan, Julie. "3,800 Civilians Killed: Unwelcome News of the 'War on Terror'." Dallas Peace Times Feb.-Mar. 2002 <www.dallaspeacecenter.org/dpt0202/civilian_deaths.htm>.
13. The Command Post. 29 Mar. 2003 <www.command-post.org/2_archives/2003_03_29. html>.
14. Beaver, Paul. "World: Europe Analysis: How Yugoslavia Hid its Tanks." BBC News 25 June 1999 <http://news.bbc.co.uk/1/hi/world/europe/377943.stm>.
15. Chow, S., P. Eisen, H. Johnson, and P.C. van Oorschot. White-Box Cryptography and an AES Implementation. Proc. of the 9th International Workshop on Selected Areas in Cryptography, 2002: 250-270. Springer-Verlag Lecture Notes in Computer Science 2003: 2595, 2003.
16. Anckaert, Bertrand, Bjorn De Sutter, and Koen De Bosschere. Software Piracy Prevention Through Diversity. Proc. of the 4th ACM Workshop on Digital Rights Management.

About the Author

Alec Main is chief technology officer at Cloakware. He has been involved in the design and implementation of numerous software security systems for PCs, mobile devices, and set-top boxes, and has pioneered Cloakware's software protection technology. Main is a regular speaker at conferences on the topic of software protection and has a degree in electrical engineering.

Cloakware
8320 Old Courthouse RD
STE 201
Vienna, VA 22182
Phone: (866) 465-4517
Fax: (703) 760-7899
E-mail: alec.main@cloakware.com