Software Protection and Application Security: Understanding the Battleground*

A. Main[1] P.C. van Oorschot[2]

[1] Cloakware Corporation, Ottawa, Canada
[2] Computer Science, Carleton University, Ottawa, Canada

Abstract. We provide a state-of-the-art explication of application security and software protection. The relationship between application security and data security, network security, and software security is discussed. Three simplified threat models for software are sketched. To better understand what attacks must be defended against in order to improve software security, we survey software attack approaches and attack tools. A simplified software security view of a software application is given, and along with illustrative examples, used to motivate a partial list of software security requirements for applications.

1 Introduction

More than 20 years ago, *data security* was defined by Denning [22] as the science and study of methods of protecting data in networked systems, and to include cryptographic controls, access controls, information flow controls, inference controls, and procedures for backup and recovery. Of these, cryptographic controls have received the greatest academic attention, with emphasis on typically mathematical data-manipulation algorithms involving secret keys – e.g. encryption algorithms for confidentiality, and message authentication codes (MACs) and digital signature algorithms for real-time authentication, data origin authentication, integrity or non-repudiation. *Applied cryptography* [39] includes additional areas of practical interest – e.g. authentication and key management protocols, and implementation issues. Complete *security infrastructures* are now used in practice, such as the Kerberos authentication service [56] and more ambitious (and correspondingly complex) key and credential management systems known as Public-Key Infrastructures (PKI) [1].

However, providing security within networked information systems goes far beyond protecting data, cryptographic keying material, and credentials. The transition from a mainframe-based computing infrastructure, through client-server architectures, to global connectivity in today's Internet has resulted in a vast array of new security threats and challenges. Indeed, it is difficult to define exactly what is meant by "security" – it is generally intended to vaguely mean protection "of valuable things" and "against bad actions".

* Dec. 31 2003. Author email: `alec.main@cloakware.com`, `paulv@scs.carleton.ca`. State of the Art and Evolution of Computer Security and Industrial Cryptography, June 2003, Heverlee, Belgium, Springer-Verlag LNCS (to appear).

More formally (e.g. see Bishop [5]), security is usually defined relative to a *security policy*, which defines actions, typically related to accessing resources (memory reads/writes, the CPU, communications ports, input-output devices, etc.), as allowed or disallowed. Methods, tools or procedures enforcing policies are called *security mechanisms*. A system is in either an allowed state (secure) or not; these states are precisely defined in theory. *Attacks* are actions which may cause *security violations* (movements to non-secure states). The security objective is to prevent, detect and/or recover from attacks.

In practice, of course, the situation is far less clear. Policies are often imprecise and incomplete common-language descriptions of what users, administrators, and outsiders are allowed to do. Typically they are neither explicitly formulated nor written down – in part due to the failure to understand the need for a security policy, and the difficulty of properly formulating one. Even experts find it challenging to accurately assess all relevant *threats* (potential violations of security) in a particular environment. Due to their large numbers and changing natures, it is virtually impossible to stay abreast of all relevant types of attacks, levels at which attacks may occur, exploitable implementation details, and complex protection mechanisms available.

In this paper, we take a step towards clarifying this picture with particular focus on application security, software protection, and software security as relevant to commercial practice. We seek to facilitate a better understanding of software protection and its intricacies, to be better positioned to improve software security. This necessitates understanding the state-of-the-art in attack tools and approaches. To this end, the sequel is organized as follows. In §2, we distinguish three architectural categories of security, all of which play a role in application security. §3 discusses practical threat models. §4 considers software attack approaches and how attacks are developed. §5 surveys software tools of use to attackers. §6 provides specific illustrative examples of where software protection is needed. §7 gives a software security view of software applications, motivating specific software security requirements. §8 ends with concluding remarks.

2 Application Security and Architectural Categories

We broadly define *application security* as the protection of software applications against threats (potentially including those not presently known). We clarify this as folows. By software applications (see also §7), we do not restrict ourselves to applications in the OSI layer-7 sense (e.g. see [54]), nor to end-user applications on client machines. In our view, application security includes, requires or depends on three inter-related *architectural categories of security*, as follows.

1. *Data security*. This is largely concerned with protecting the confidentiality and integrity of data, typically in transit and storage (cf. Denning's definition above). We note that a typical assumption in data security and cryptography, that end-points are trusted, does not generally hold in environments requiring application security (see §3).

2. *Network security.* We view this as the science and study of the protection (including access to, availability and integrity) of network resources, devices and services (including bandwidth, connectivity, and platforms). We include, within network security mechanisms: firewalls, intrusion detection systems (IDS), systems providing access control to devices, and mechanisms to ameliorate denial of service (DOS) and distributed DOS (DDOS) attacks.
3. *Software security.* We define this as the science and study of protecting software (including data in software) against unauthorized access, modification, analysis or exploitation (cf. [62, 33, 60]; see also *program security* [5, Ch.29]). We also use the term "software security" informally (see discussion below) in relation to the security properties and level of inherent security in a software application (in the sense of protection against relevant attacks).

While *software protection* is a term sometimes interchanged with *copy protection* (e.g. [27, 29]), our definition differs. In our view, software protection consists of a broad collection of principles, approaches and techniques intended to improve software security, providing increased protection against threats ranging from buffer overflow attacks [66] to reverse engineering and tampering [20]. We thus view copy protection as only one application of software protection; other example end-goals are *preventing unauthorized use* of software, and *content protection*: preventing unauthorized use of bulk data processed by software, e.g. music files, streaming audio or video. Here *unauthorized use* means uses other than intended by the legitimate software creator.

We distinguish software protection from *security code*, which provides narrower or more localized security-specific functionality. Security code is a part of the application, contributing security functionality such as authentication (e.g. for copy protection) to a user, server, machine or other hardware such as a CD. (Network security software – including virus scanners, firewalls, intrusion detection systems and other perimeter-type defenses – consists largely of security code, but in applications unto themselves.)

More specifically, in our view (cf. [57]), applications consist of three types of software: functional code (which accomplishes the primary goal), error-handling code, and security code. In contrast to the latter, we see software protection as binding together (reinforcing) the existing software, changing its inherent structure or properties – analogous to adding cement to sand, gravel and water to make a monolithic slab from three otherwise easily separable components. While security code plays an important role in the overall security design, it may itself be subject to attack; software protection ideally makes the software itself resistant to attack.

In summary, software security is dependent on both security code and software protection. Furthermore, depending on the threat model (§3), application code other than security code may also require software protection.

Many software protection techniques and approaches exist. Some can be classified into major groups: software obfuscation, software tamper resistance, diversity, marking schemes (e.g. watermarking), node-locking schemes, time-limiting schemes, etc. (see also §7). For further details, see recent surveys [4, 20, 60].

3 Threat Models for Software

An enormous number of security threats to software-based systems exist. In reality, no system can be 100% guaranteed to be "totally secure"; our knowledge of attacks is incomplete, and new attacks are devised regularly. Even in the best case, we are aware of only a subset of possible threats at any point in time, and of these, not all warrant defenses – it is too costly to design a system to be secure against every known threat. Some have near zero likelihood of being mounted or succeeding; others would result in relatively minor losses if they did succeed, or are too costly for an attacker to mount despite being theoretically possible. The goal in practice is to cost-effectively counter relevant threats.

3.1 What is a Threat Model?

A *threat model* identifies the threats a system is designed to counter, taking into account the nature of relevant classes of attackers (including their expected attack approaches and resources – e.g. techniques, tools, powers, geographic access), as well as other environmental assumptions and conditions. Pertinent questions include: "Which resources need protection, and from which specific malicious actions?" Answering such questions requires *risk assessment* (cf. [9]): analyzing threats and system vulnerabilities; estimating potential losses; and estimating the likelihoods of particular attacks and their success. The threat model should be used within the system design process, and a *security model* should then be devised showing how the system design counters relevant threats.

As should now be clear, threat models are based on assumptions (which are not always correct), environmental conditions (which may change over time), imperfect judgements, and unconfirmable estimates. The quality of a threat model for a software application depends on how closely it reflects the reality of the environment in which the application runs, and the threats it is actually subjected to. Far from an exact science, this is at best an approximate art, and almost always an interative process adapting to better knowledge, new threats and changing conditions.

As a case in point of how threat models may not accurately reflect reality, in a section titled *The Internet Threat Model*, Rescorla [49, p.1] states two major assumptions around which most Internet security protocols including SSL are built: (1) the protocol end-points (e.g. client and server machine) are secure; and (2) the communications link is totally vulnerable to attacker control (e.g. eavesdropping, message modification, message injection). This follows the typical cryptographer's model, which is suitable for securing data transmitted over unsecured links in a hostile environment. However, it is clear that assumption (1) is no longer generally valid for the Internet circa 2003, with large-scale malicious software [51] compromising the integrity of vast numbers of machines (e.g. 350 000 machines due to a Code Red worm circa July 2001 [7]). As noted earlier, in many places where application security is required, this standard cryptographic *trusted endpoint assumption* fails. It is of course well known that cryptography alone cannot solve all software security problems; for example, a 1998 U.S. National

Research Council report [52, p.110] notes that less than 15% of all problems in CERT advisories are cryptographically addressable.

3.2 Three Basic Threat Models (Classes of Attacks)

An overall threat model may incorporate any or all of the following sketches.

a) *Network Threat Model.*
An increasing number of software applications are *network-connected* – e.g. browsers, mail clients, word processing applications and spreadsheets. These are both willing and able to provide functions across a network to other network-accessible applications. Such applications are vulnerable to remote external attacks, as they can be caused to directly process data from remote processes. Here the target application is assumed to reside on a host machine (hardware plus software) which is both trustworthy and under control of a trustworthy entity (e.g. the individual running the application, or an entity hosting application services). Typical software security concerns include: buffer overflow exploits [66]; protocol attacks exploiting "normal" features of a protocol (e.g. [11, 6]); and malicious code [35, 65, 58] that allows unauthorized access and/or intrusion on the application, data and hardware by an outsider. Attacker goals typically include gaining privileged access to the machine to carry out other malicious actions, e.g. launching other attacks, consuming resources, or harvesting information.

b) *Insider Threat Model.*
Here the attacker has some level of privileges on either the network (e.g. inside a corporate firewall, on a local area network) or hardware running the target application. Software security concerns include software piracy, local exploits to boost privileges, and stealing information embedded in or processed by the software. The attacker's goal is typically to gain privileged access to the device, or to steal or tamper with data or applications.

c) *Untrusted Host Threat Model.*
Under the assumption of an application running on an untrusted host machine [50], the application is subject to attacks originating from the host machine itself – e.g. the operating system, kernel, other applications, the hardware, etc. The attacker is assumed to be local and to enjoy full privileges, including unconstrained access to the running code, and installation and use of additional software or hardware attack tools. This *white-box attack context* requires special software protection techniques [13], and differs from both a *black-box attack model* (where the attacker can only monitor inputs and outputs), and a *side-channel attack model* (where the attacker can monitor side effects of program execution). Software security concerns include: piracy of data and applications, unlicensed use and tampering of data or applications, and theft of software-embedded information and intellectual property. Attacker goals include: "freeing content" (removing any constraints or controls thereon), freeing the application (removing copy protection or the requirement for license strings), information theft, and unlicensed use of the data or application.

	Network Threat Model	Insider Threat Model	Untrusted Host Threat Model
privileges of attacker	none	some	full
attacker location	remote	local network or same host	same host
host trusted?	yes	yes	no

Table 1. Comparison of selected characteristics of three threat models.

Note: as successful network and insider attacks may lead to compromise of the target machine, these attacks may evolve to create an untrusted host scenario (see related remarks in §8).

Table 1 summarizes a few of the major differences between these threat models. Other distinguishing characteristics include: types of tools available, prevalent types of attacks, and goals of attackers (which map directly to software security concerns). Providing application security under each of the threat models of Table 1 generally requires the use of security mechanisms from all three architectural categories: data security, network security, and software security. One exception is in the untrusted host model: there, network security protection is less relevant, and standard data security mechanisms provide only static protection of data.

4 Software Attacks

Attackers may achieve their goals through many approaches. §4.2 discusses direct access attacks. §4.3 addresses automated attacks. §4.4 reviews the steps in developing large-scale attacks. We begin with a note regarding terminology.

4.1 Terminology: Hackers, Crackers and Hats

In line with security researchers, we refer to someone trying to defeat security policy or mechanisms as an *attacker*. This is meant as an unbiased term – the attack motives may be honorable (e.g. penetration testing by product developers or researchers to find vulnerabilities in order to fix them), or not (e.g. breaking in for malicious reasons or "fun").

Following the computer technical community (cf. [48, 47, 28]), we reserve the term *hacker* for computer gurus familiar with low level programming details and how to "stretch" programming functionality. We view it to have a generally positive ("geeky" to some) connotation. In contrast, we reserve the term *cracker* for individuals who break into computer systems, typically bypassing or defeating security mechanisms. Hackers generally take pride in having ethical standards, and view crackers as vandals or thieves who misuse their technical skills, including to gain unauthorized access. The mass media most often uses the term "hacker" in place of "cracker"; we find this confusing, and in what follows intend the terms as defined above.

Several related terms are worth mentioning. *Ethical hacking* is used as a term for the practice, by those with sufficient skills and with the advance permission of the system owners, of breaking into computer systems to demonstrate security weaknesses. Two additional terms stem by analogy from Western movies, where "good guys" wear white hats and "bad guys" wear black. The term *white-hat* (white hat hacker), having a positive connotation, is associated with those using their skills for legitimate purposes, e.g. computer security experts doing system research or vulnerability testing to better defend against attacks. In contrast, the term *black-hat*, having a negative connotation, denotes unauthorized individuals who break in to computer systems for illegitimate purposes – thus being synonomous with crackers. Finally, *grey-hat* (gray-hat) denotes white-hats who sometimes go beyond what white hats typically do.

4.2 Direct Access Attacks

Analysis and tampering typically involve direct access to the target code, involving a skilled attacker, with sufficient resources and time to manipulate the code in a controlled environment. We usually define *direct access attacks* to be those developed on a local machine using a local copy of the target code. However in some cases, the term may also include attacks developed over a network connection; the main point is direct human involvement. Note that under some threat models, attacker access to the code and environment is straightforward.

We now consider the differences between software analysis and tampering, and static vs. dynamic approaches.

i) *Analysis vs. Tampering.* While software reverse engineering (see §5.1) may be an attacker's end-goal, often it is a precursor to tampering, which requires that the attacker first sufficiently understand the internals of an application. When reverse engineering leads to discovery of an exploit or vulnerability, the code may then be modified to perform as the attacker desires.
ii) *Analysis: Static vs. Dynamic.* Static analysis refers to analysis of software and data when it is not running. It typically includes static disassembly (see §5.3) of executable code and subsequent examination. In contrast, dynamic analysis, performed on executing code, involves tracing of data values and control flow. While typically more powerful than static analysis, it may be more time consuming, more complicated, and requires a platform similar to that of the target code.
iii) *Tampering: Static vs. Dynamic.* Software tampering attacks may similarly be static or dynamic. A static tampering attack modifies code in a non-executing state; the modified code is subsequently run. If a software integrity mechanism is in place (e.g. the operating system verifies integrity using code-signing techniques, or the code checks itself [36, 12, 32]), then the integrity-checking mechanism must be defeated for the modified software to execute as desired. A dynamic tampering attack changes values (data or code) in memory during execution. An attack may be developed or tested dynamically, on a separate platform, and then turned into a static attack on

a target platform. A typical goal of tampering attacks is software piracy, or unauthorized duplication of files in violation of a licensing agreement.

Many more complex direct access attacks exist. We mention of few here which have analogies in data security and cryptography.

Software differential analysis (SDA) is a powerful attack that one or multiple attackers may use. Two or more different versions of an application are compared, to identify which parts have been changed – e.g. between product releases (over time), or between users (per-user variations). Crackers who develop copy protection removal tools use SDA to quickly isolate changed protection techniques, updating their tools to allow them to continue working on new releases.

Collusion attacks involve multiple attackers sharing analysis, to leverage not only different skills to reverse engineer a system, but also to pool user-specific data or knowledge of help in defeating security mechanisms (see SDA above).

Replay attacks capture program state and later restore it. For example, a user downloading a movie may watch it within three days of pressing play (e.g. movielink.com). A backup of the entire machine state is made using a disk imaging tool (e.g. www.powerquest.com/driveimage/). Once the original digital rights are consumed, the user restores them using the back-up machine state.

4.3 Automated Attacks

Once a direct access attack has yielded an exploit proven to work, often the attacker's goal is to automate the attack so that it can be rapidly deployed on a broader scale, including by others. Internet viruses and worms are evidence of the power of automated attacks. Viruses consist of three parts: exploit, vector, and payload. The exploit, taking advantage of a discovered or known vulnerability, is first developed as a direct access attack. The vector is the means of self-propagation, providing automation. The payload code does the damage. Automated attacks also apply to copy protection schemes, such as CSS used with DVDs [23, 42], and unSafeDisc, a software tool used to remove (Macrovision Corporation's) SafeDisc copy protection from PC games.

Automated attacks have limitations. They typically necessarily involve numerous implementation assumptions regarding the target code. Special markers and offsets are used to know where to apply a patch or inject values. If a specific target code instance differs slightly in critical places – e.g. due to a slightly different product release – the automated attack will fail on that instance. For this reason, highly automated attacks are often *fragile*. Automated attacks are also typically limited in analysis, and do not generally incorporate feedback. Nonetheless, automated attacks are highly effective in today's Internet, due to the homogeneity of the installed base of applications.

4.4 How Software Attacks are Developed

Software attacks often proceed according to the following sequence.

1. *Analysis.* This is typically the first step in an attack, although in some cases it may be the end-goal – such as in theft of intellectual property (e.g. to understand a communications algorithm).
2. *Tampering.* This involves modifying the code and/or data to perform other than originally intended. Some attacks involve tampering only; typically tampering occurs after some analysis.
3. *Automation.* This typically involves the development and use of software tools to create and apply attack software to multiple instances of the target application. In some cases the tampered application is re-distributed (e.g. direct piracy of copy-protected PC games). With content protection schemes, preference is for a general automated attack that works for all content protected with that scheme (e.g. DVD CSS [23, 42]). In the case of computer viruses or worms, the attack goal is to obtain privileges, cause damage, or consume resources on many devices.
4. *Distribution.* This allows others to use the automated attack. Attacks may also be manually distributed via web sites and bulletin boards. Some attacks are highly prized by groups and not distributed. Computer worms include mechanisms for self-propagation.

5 Software Attack Tools

Like many areas of security, and perhaps moreso than most, software security is an arms race between those who create and deploy security mechanisms, and attackers eager to defeat them. In order to provide stronger protection, or (more pessimistically) at least understand why many of the present protection techniques are inadequate, it is thus essential to understand the techniques attackers use to defeat them. To this end, this section briefly discusses software reverse engineering, attack techniques, and attack tools.

5.1 Software Reverse Engineering

Software reverse engineering – given an application's binary code, analyzing the program and recovering the details within it – is an open-ended research area, for both those interested in software protection, and in defeating it (see [45, 24] for different views and resources). Major aspects of reverse engineering include *disassembly* and *decompilation*. Decompilation recovers higher-level program abstractions and semantic structure from binary programs. Disassembly reconstructs assembly language instructions from machine code; it may be considered a subset of decompilation, or a step along the way. Reverse engineering of *type safe* languages such as Java is much easier than most other languages, like C and C++, whose binaries contain no type information, and whose executables have program code and data mixed within their runtime layout [63].

Attackers become aware of security vulnerabilities through many means, including: announcements by product vendors typically accompanied by patches (attackers then target unpatched installations); similar announcements by independent researchers (without patches); and independent discovery. Reverse engineering for malicious purpose – e.g. theft of intellectual property (such as a competitor's secret formula or process), software tampering, or the discovery and exploitation of vulnerabilities – is facilitated by a number of advanced program analysis tools which also serve the legitimate software development community, e.g. in debugging, software engineering, and understanding malware. In both cases, useful capabilities for reverse engineering include:

1. observation of dynamic program behavior in a controlled manner, including granularity down to instruction-by-instruction execution;
2. recovery, from binary, of assembler code and higher-level abstractions; and
3. dynamic modification of binary code and observation of resulting behavior.

To this end, foundational tools in the cracker's reverse engineering toolkit include: debuggers, disassemblers, decompilers and emulators. Each of these are discussed in turn below, as well as additional attack tools.

5.2 Debuggers

Debuggers (e.g. see [17]) trace the program logic and data values during program execution. Breakpoints can be set and code and data modified on the fly, making debuggers valuable tools for uncovering bugs and addressing performance issues, as well as reverse engineering and tampering with applications. *SoftICE* [40] is powerful debugger (software in-circuit emulator) for Microsoft Windows environments and Pentium family processors; crackers write plug-ins to extend its functionality, and to provide information for widely hacked applications. Another debugger for Windows environments is the free *OllyDbg* [41]. The GNU Project debugger, *GDB* [25, 55], is the most popular debugger for Unix systems.

5.3 Disassemblers

A disassembler is typically the first tool used in reverse engineering an executable program, whether for legitimate purposes (e.g. automated code optimization) or otherwise. Disassemblers allow analysis of binary code, mapping it to assembly language (additional decompilation steps may allow recovery of higher-level constructs). Code navigation may be simplified by identifying API calls, building and displaying call graphs, and producing anotated assembly-language listings. For extensive background and pointers to online tools, see the *Disassembly* pages at the Program Tranformation Wiki [46]; for a recent research paper on disassemblers, see e.g. Schwarz et al. [53].

IDA Pro [21], viewed by many as the most sophisticated commercially available disassembler for C, supports essentially all processors on the market; it is also the only one capable of binary reverse engineering C++ code [63]. IDA Pro is an *interactive disassembler* – it relies on human intervention to control which parts of the target binary to disassemble (it is actually both disassembler and debugger). Disassembly capabilities exist in many other tools. An example of a free tool is the GNU Project *objdump* utility [26]; it also has capabilities to display embedded debugging information.

Techniques to disrupt the process of static disassembly of programs have recently been explored by Linn and Debray [37] (see also Cohen [19]). The goal is to make correct disassembly more difficult. Their techniques are complementary and orthogonal to software obfuscation. (Note: in *static disassembly*, a target binary is disassembled without executing the code; the complete file is typically disassembled. In *dynamic disassembly*, program execution is monitored by an additional tool, typically a debugger, with instructions identified as executed; the executed program "slice" is diassembled.)

5.4 Decompilers

Machine code decompilation has a long history, but few commercial tools exist for non-type-safe languages like C and C++ [14, 15, 8]. For detailed background, see the *History of Decompilation* at the Program Transformation Wiki [46]. The most advanced decompilation tool for C is currently the open source University of Queensland (Australia) Binary Translator, UQBT [16, 18], albeit developed for binary translation; see Vinciguerra et al. [63] for a brief summary of capabilities.

Decompilation of intermediate level code intended to run on a virtual machine (e.g. Java bytecode or Microsoft's C# CLR – Common Language Runtime) is much easier, and a number of commercial and freeware decompilers exist – e.g. *SourceAgain* [2], a decompiler for Java class files. Obfuscators for these languages, such as *Dotfuscator* [44], a Microsoft .Net obfuscator and compactor, typically focus on program optimization by reducing symbol names, but do not prevent serious reverse-engineering.

5.5 Emulators, Simulators and Spoofing Attacks

Emulation and *spoofing attacks* are methods that, rather than tampering directly with an application, exploit an interface or impersonate presumably-trusted system components. As one example, an expensive CAD program is protected with a dongle that responds to challenges from the application to prove the dongle's presence. After reverse engineering the interface and response mechanism, an attacker replaces the dongle's driver with a tampered driver providing the same interface but with an emulated dongle. This resembles a (dynamic) replay attack.

Thus emulators and simulators (e.g. [38, 64, 43]) allow crackers to emulate the environment in which an application expects to run. Emulators can be used to store state information (as may debuggers), to help replay attacks (cf. above). They are also used to create virtual drives to bypass copy protection schemes – for example, *Alcohol 120%* [3] makes 1:1 CD/DVD copies and allows emulation.

We note that from a defensive stance, determining whether or not an emulator is running is a challenging problem [31].

5.6 Other Software Attack Tools

Beyond the tools noted above, various others are commonly used. *Anti-debug techniques* (see e.g. [62, pp.416-418], [10]), used for software protection and by computer virus writers, are typically defeated by customized tools. Memory dump and memory lift utilities are available, including for binaries that are packaged with compression or encryption wrappers. For example, *DumpPE* dumps internal structures of Windows95 and NT PE or Portable Executable files (e.g. .exe, .dll, .dbg); the *PE Explorer* tool [30] unpacks, disassembles, analyzes, and edits PE files. *ProcDump* is a popular Win32 unpacker, for capturing data from arbitrary memory locations. Hex editors such as *Hackman* [59] are used to piece code together or modify executables. *FileMon* [67] is a utility which monitors and displays real-time file system activity, e.g. allowing a view of how applications use files and DLLs (for Windows environments; also a version for Linux). Most of these tools are available through legitimate channels (free or at reasonable cost); pirated versions and specific cracking tools are available on many cracker web sites, bulletin boards and online networks.

6 Software Protection Examples

This section provides illustrative examples for software protection. These partially motivate the software security requirements listed in §7. For additional examples, see e.g. D'Anna et al. [4, §2.2].

A) *Network Services Attack.*
A network application is attacked by a worm exploiting a buffer overflow vulnerability to install a backdoor on the host device. Subsequently, a cracker penetrates the network and has full privileges on the (now untrusted) host.
Comments: This is a remote, dynamic tampering attack on the input software in the application. Software protection techniques should be used to eliminate the buffer overflow vulnerability (e.g. by proper validation of application input data).

B) *Secret Algorithm Protection.*
In a highly secure environment at a department of defense signal processing center, a rogue employee copies a critical application onto a floppy disk or USB token, sneaks it outside the facility and sells it to a foreign agent. In a foreign lab (with the application now on an untrusted host), the application is analyzed to recover algorithms to allow their use in competitive systems, or to gain knowledge to defeat the original mechanisms.
Comments: The internal data and algorithms require additional software protection. This insider attack has certain implications for software security: since the attacker may not be able to replicate the exact platform that the application runs on, dynamic analysis might not be applicable. A possible defense is a cryptographic wrapper that decrypts the application upon entry of a valid password or machine identifier.

C) *CD Copy Protection.*
A PC game publisher, selling games on CDs, intends that a user may install a game on many machines, but can only play it when a valid CD (one bought, vs. created by copying) is in the drive. Security code in the game software checks for CD validity, by various authentication techniques exploiting differences between (a) how CD drives read CDs, and (b) how copies are burned. One recent technique measures timing differences reading certain tracks (timings are the same for all CDs stamped from the same glass master, but glass masters differ slightly; that used for a blank CD differs from that used for the game). The authentication mechanism compares a value from the measurement algorithm (security code) with a known value (data).
Comments: The game resides on an untrusted host. Software protection is required to protect the CD authentication code from being bypassed. The attacker may try to alter decision making code to always return "valid"; or reverse engineer the measurement algorithm in order to (a) emulate it, or (b) produce a new valid value for the copied CD.

D) *Data Security on a Server.*
An insider steals an encrypted database of credit card information stored on a server, as well as the application used to access the database. The application code contains the cryptographic key to decrypt the data. (The attacker wishes to recover the credit card information but it was too difficult to do this dynamically on the server.) The attacker statically analyzes the application to recover the key used to encrypt the database, then writes a utility to decrypt the database.
Comments: The database uses data security to ensure confidentiality of the information. The application's input software however must decrypt the data and manipulate it. The keying material used by the input software should be protected to hide the values (as a minimum, from static analysis).

E) *Dynamic Data Protection.*
A user installs a tool in their directory on a server, scanning memory for RSA private keys, easily identified by their size and randomness [61]. Candidate keys are stored in a log file and verified using public key certificates. Once the server's private key is confirmed, the server can be spoofed.
Comments: The application needs software protection, including to prevent dynamic analysis of keying material.

F) *DVD Content Protection.*
Video data on a DVD is encrypted with the Content Scrambling System (CSS) algorithm [23, 42]. When DVDs were originally launched, CSS and the keys used were secret. The intent was that only DVD players which knew the algorithm and had one of the correct keys could decrypt data. One company implemented the algorithm (security code) and a valid DVD key (data) in an unprotected application, which ran on an untrusted host (a PC). Both the algorithm and secret key were reverse engineered and extracted by crackers. Others were then able to write their own software to decrypt DVDs.
Comments: Software protection should have been applied to the security code to provide algorithm and data hiding.

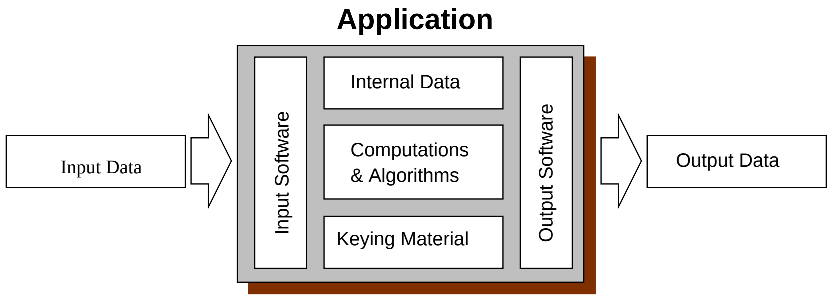

Fig. 1. Simplified Software Security View of a Generic Application

G) *Digital Rights Management (DRM).*
A DRM system allows users to download digital content to a PC and play it for a limited time. By design, a DRM server only sends data to valid players; when a request is made, the player must authenticate itself (security code). This sets up a server-player session key to ensure the encrypted content is not intercepted in transit (data security). The server checks if the player has been revoked before sending content. The player has protected the authentication keying material using software protection. The player also checks itself for tampering, and certain system components which (for performance constraints) are not tamper-resistant. On detecting tampering, it alerts the server during the content request. Revocation or tampering detection may require user software upgrade [34]. Content played is decrypted using the content keys, rendered and e.g. displayed on a screen output. A watermark is added and the content resolution slightly degraded during rendering.
Comments: Software protection is needed to protect the rendering algorithm, and to hide content keys, decrypted content, and the watermark application. Rendered content is not hidden, but considered lower value to an attacker. The player must be tamper resistant to prevent "siphoning off" content.

7 Software Security: Application View and Requirements

A simplified view is that an application is software that runs on hardware and manipulates data. A more realistic picture is that an application relies on an operating system which in turn relies on a kernel and device drivers to communicate with the hardware. The application may be distributed across various servers and clients that communicate across a network. Thus in the broadest sense, all the software, hardware and data that participate are part of the application, and are thus involved in application security. Here we will view all software, including system and operating system software, as an application, and include software running on all types of devices – not only end-user personal computers, personal communications devices and personal digital assistants, but also servers, switches, firewalls, routers, and other network devices.

The simplified model of Figure 1 represents an application as a software process that inputs data, manipulates it and outputs data. Software involved in input-output processing is distinguished from the data itself. In a hostile environment, an application's input-output data may ideally be protected by data security techniques; the software processing such data may be hardened by software protection techniques. The model separates internal data into *keying material* and other internal data. To help define software security requirements (below), the following application components are high-lighted.

1. *Input Software.* Software that reads data into an internal representation for processing. If the input is cryptographically protected, the input software functionality includes data decryption, integrity verification, etc.
2. *Output Software.* Software that writes data to an output medium (e.g. RAM, CD, socket) after processing. If the output data is cryptographically protected, then output software functionality includes data encryption, integrity processing, etc. Applications are often chains of programs, but typically at some point *some* data must be output in the "clear", such as to a screen.
3. *Internal Data.* Data initialized by the application, data read into an internal representation, or data computed within the application (e.g. intermediate values or in preparation for output).
4. *Keying Material.* A subset of internal data that is typically of high value to an attacker. Among other things, this may include cryptographic keys (e.g. which must remain private or whose integrity must be guaranteed), security-critical data and privilege-related data.
5. *Computations and Algorithms.* The internal logic embodied in the program that processes the internal data and keying material.

The above view, and §6 examples, motivate the following, which we propose as a partial list of high-level software security requirements for applications. Its satisfication requires software protection and may also require security code.

1. *Securing the input.* Authentication, validation, hiding and integrity of data input to an application. (Example approaches: PKI, bounds checking, type checking, digital signatures, check sums, white-box cryptography.)
2. *Securing the output.* Authentication and hiding of an application's data output. (Example approaches: similar to those for securing the input.)
3. *Data hiding.* Hiding data and keying material from direct access (§4.2) static or dynamic analysis. (Example approaches: obfuscation, code transformations, wrappers, just-in-time decryption, self-modifying code, techniques thwarting analysis tools e.g. anti-debugger techniques.)
4. *Algorithm hiding.* Hiding algorithms and computations from direct access (§4.2) static or dynamic analysis. (Example approaches: see data hiding.)
5. *Tamper resistance.* Making software difficult to modify or tamper. This may include static and dynamic *tamper detection* i.e. detecting integrity violations of any component of a software application or its operating environment. (Example approaches: many of those listed for data hiding above; code-signing and dynamic self-checking techniques.)

6. *Damage mitigation.* Proactive and reactive strategies for protecting the application infrastructure or installed base once an application is compromised. (Example approaches: revocation, renewability, software diversity.)

8 Concluding Remarks

Most attacks in today's Internet environment exploit software. Many of these result in an attacker gaining control of an application's execution environment. This brings the untrusted host threat model (§3) into relevance, leading to two observations. 1) The threats inherent in this model must be addressed in the security models that should be part of the software design process for applications. 2) Software protection deserves a higher priority in the software industry. Applications consisting solely of functional code, error-handling code, and security code (§2), without software protection to harden these code groups, typically falls short on the security level required for safe, reliable software-based systems.

We believe that the software industry, and thus our industrial world (relying heavily on software-based systems), is in a dangerously precarious state due to the ease with which attackers exploit software vulnerabilities and tamper with software-based systems. The software industry, its security experts, and the academic research community, are losing ground in the battle against attackers of computer software systems. New copy protection schemes are broken shortly after their commercial introduction; new watermarking schemes are quickly broken by crackers; buffer overflow attacks continue to dominate lists of reported computer software incidents; and the damage caused by computer worms and viruses continues to increase each year. We believe that there is much to be learned from the cracker community. The tools and resources available online, freely shared and at the disposal of crackers, is both astounding and worrisome, as is the number of pirated software applications and intellectual property (including movies and music). This calls for greater attention to research in software security – in both industry and academia – and greater use of available software protection technologies. This presents a tremendous opportunity for security researchers.

Software security is a highly interdisciplinary field, involving experts from diverse areas of computing science and engineering including: programming languages, operating systems, compilers, software engineering, network security and cryptography. We expect that over the next 10 years, many advances will come from researchers who successfully span several of these fields.

Acknowledgements. The second author acknowledges Canada's National Sciences and Engineering Research Council (NSERC) for funding an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security.

References

1. C. Adams, S. Lloyd, *Understanding Public-Key Infrastructure* (2nd edition), Macmillan Technical, 2002.

2. Ahpah Software Inc. (Moutainview, California), http://www.ahpah.com/.
3. Alcohol Soft Development Team, http://www.alcohol-soft.com/.
4. L. D'Anna, B. Matt, A. Reisse, T. Van Vleck, S. Schwab, P. LeBlanc, "Self-Protecting Mobile Agents Obfuscation Report", Network Associates Labs Report #03-015, 30 June 2003.
5. M. Bishop, *Computer Security: Art and Science*, Addison Wesley, 2003.
6. D. Bleichenbacher, "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", *Advances in Cryptology – CRYPTO'98*, Springer-Verlag LNCS 1462 (1998), pp.1–12.
7. CAIDA (Cooperative Association of Internet Data Analysis), CAIDA Analysis of Code-Red, http://www.caida.org/analysis/security/code-red/, visited 5 Dec. 2003.
8. G. Caprino, http://www.backerstreet.com/rec/rec.htm/, "REC - Reverse Engineering Compiler", accessed 29 December 2003.
9. CMU Software Eng. Inst., Definition of Software Risk Management, accessed 5 Dec. 2003, http://www.sei.cmu.edu/programs/sepm/risk/definition.html.
10. S. Cesare, "Linux Anti-debugging Techniques (Fooling the Debugger)", Jan. 1999, http://www.uebi.net/silvio/linux-anti-debugging.txt, accessed 29 Dec. 2003.
11. CERT Advisory CA-1996-26 Denial-of-Service Attack via ping ("Ping of Death"), http://www.cert.org/advisories/CA-1996-26.html, accessed 29 Dec. 2003.
12. H. Chang, M. Atallah, "Protecting Software Code by Guards", pp.160–175, Proc. 1st ACM Workshop on DRM (DRM 2001), Springer-Verlag LNCS 2320 (2002).
13. S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, "White-Box Cryptography and an AES Implementation", Proc. 9th International Workshop on Selected Areas in Cryptography (SAC 2002), Springer LNCS 2595 (2003).
14. C. Cifuentes, *Reverse Compilation Techniques*, Ph.D. thesis, Queensland University of Technology (Australia), Dept. of Computing Science, 1994.
15. C. Cifuentes, K.J. Gough, "Decompilation of Binary Programs", *Software – Practice and Experience*, vol.25 no.7 (July 1995), pp.811-829.
16. C. Cifuentes, M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost", *IEEE Computer* vol.33 no.3 (March 2000), pp.60-66.
17. C. Cifuentes, T. Waddington, M. Van Emmerik, "Computer Security Analysis through Decompilation and High-Level Debugging", Workshop on Decompilation Techniques, pp.375-380, 8th IEEE WCRE (Working Conf. Rev. Eng.), Oct.2001.
18. C. Cifuentes, M. Van Emmerik, N. Ramsey, B. Lewis, "Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework", Jan.2002, http://research.sun.com/techrep/2002/smli_tr-2002-105.pdf.
19. F. Cohen, "Operating System Protection Through Program Evolution", *Computers and Security* vol.12 no.6, 1 Oct. 1993, pp. 565–584.
20. C.S. Collberg, C. Thomborson, "Watermarking, Tamper-Proofing, Obfuscation - Tools for Software Protection", *IEEE Trans. Soft. Eng.*, vol.28 no.6 (June 2002).
21. DataRescue Inc. (Liège, Belgium), IDA Pro 4.6 Disassembler and Debugger, site accessed 29 Dec. 2003, http://www.datarescue.com/idabase/.
22. D. Denning, *Cryptography and Data Security*, Addison Wesley, 1982.
23. Ed Felton, 28 November 2001, Declaration in Support of Motion for Summary Judgement, in DVDCCA v. McLaughlin, Bunner, et al., Case No. CV–786804, Superior Court of the State of California, County of Santa Clara, U.S.A., http://www.eff.org/IP/Video/DVDCCA_case/20011128_felten_decl.html.
24. Fravia's pages of reverse engineering (cracker's viewpoint), site accessed 29 December 2003, http://fravia.anticrack.de/.
25. GNU Project, GDB: The GNU Project Debugger, site accessed 29 December 2003, http://www.gnu.org/software/gdb/documentation/.

26. GNU Project, *objdump* binary utility, site accessed 29 December 2003, http://www.gnu.org/software/binutils/manual/html_node/binutils_toc.html/.
27. O. Goldreich, R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs", *Journal of the ACM*, vol.43 no.3 (May 1996), pp.431–473.
28. V. Golubev, "White, Grey and Black Hackers Hats", 28 Jun. 2003, http://www.crime-research.org/eng/news/2003/06/Mess2803.html, accessed 29 Dec. 2003, Computer Crime Research Center web site.
29. J. Gosler, "Software Protection: Myth or Reality?", *Advances in Cryptology – CRYPTO'85,* Springer-Verlag LNCS 218 (1985), pp.140–157.
30. Heaventools Software (Vancouver, B.C., Canada), *PE Explorer* utility, site accessed 29 Dec. 2003, http://www.heaventools.com/.
31. R. Kennell, L.H. Jamieson, "Establishing the Genuity of Remote Computer Systems", Proc. of 12th USENIX Security Symposium (August 2003), pp.295-310.
32. B. Horne, L. Matheson, C. Sheehan, R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance", pp.141–159, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320 (2002).
33. M. Howard, D.C. LeBlanc, *Writing Secure Code*, 2nd ed., Microsoft Press, 2002.
34. M. Jakobsson, M.K. Reiter, "Discouraging Software Piracy Using Software Aging", pp.1–12, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer-Verlag LNCS 2320 (2002).
35. D.M. Kienzle, M.C. Elder, "Recent Worms: A Survey and Trends", pp.1-10 in [51].
36. G.H. Kim, E.H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker" (Feb. 1995), pp.18–29, Proc. 2nd ACM Conference on Computer and Communications Security (1994).
37. C. Linn, S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", pp.290-299, Proc. 10th ACM Conference on Computer and Communications Security (ACM CCS 2003), Wash. D.C., Oct. 2003 (ACM Press).
38. P.S. Magnusson, M. Christianson, J. Eskilson et al., "Simics: A full system simulation platform", *IEEE Computer* vol.35 no.2 (Feb.2002), pp.50-58.
39. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996 (5th printing 2001, with corrections), full text at http://www.cacr.math.uwaterloo.ca/hac/.
40. NuMega (Braeside, Victoria, Australia), SoftICE 4.0 kernel mode debugger, http://www.microway.com.au/compuware/softice.stm, accessed 29 Dec. 2003.
41. O. Yuschuk (Olly), *OllyDbg* analysing debugger for MSFT Windows (binary code to assembler), http://home.t-online.de/home/Ollydbg/, accessed 29 Dec. 2003.
42. Openlaw DVD/DeCSS Forum, Frequently Asked Questions (FAQ), accessed 29 Dec. 2003, http://cyber.law.harvard.edu/openlaw/DVD/dvd-discuss-faq.html.
43. plex86.org, Plex86 x86 Virtual Machine Project, http://plex86.org, site accessed 29 December 2003.
44. Preemptive Solutions Inc. (Cleveland, Ohio), *Dotfuscator*, site accessed 29 Dec. 2003, http://www.preemptive.com/.
45. Program-Transformation.Org, Reengineering Wiki, http://www.program-transformation.org/twiki/bin/view/Transform/ReengineeringWiki, site accessed 29 December 2003.
46. Program-Transformation.Org, The Program Transformation Wiki, De-Compilation page, site accessed 29 December 2003, http://www.program-transformation.org/twiki/bin/view/Transform/DeCompilation/.
47. Red Hat, Inc. (2002), *Red Hat Linux 9: Red Hat Linux Security Guide*, Chapter 2 – Attackers and Vulnerabilities, http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/security-guide/, accessed 29 December 2003.

48. E.S. Raymond, Jargon File version 4.4.6, http://www.jargon.8hz.com/, accessed 29 December 2003. (See also print version: Eric S. Raymond (ed.), *The New Hacker's Dictionary*, third edition, 1996, MIT Press.)
49. E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison Wesley, 2001.
50. T. Sander, C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", pp.44–60, *Mobile Agents and Security*, G. Vigna (ed.), Springer LNCS 1419 (1998).
51. S. Savage, ed., *Proceedings of the 2003 ACM Workshop on Rapid Malcode* (WORM'03), Washington D.C., 27 October 2003, ACM Press.
52. F. Schneider (ed.), *Trust in Cyberspace*, report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, (U.S.) National Research Council (National Academy Press, 1999).
53. B. Schwarz, S. Debray, G. Andrews, "Disassembly of Executable Code Revisited", pp.45–54, Proc. 9th IEEE WCRE (Working Conference on Reverse Engineering), Nov.2002, Richmond, VA.
54. W. Stallings, *Data and Computer Communications*, 7th ed., Prentice Hall, 2004.
55. R.M. Stallman, R.H. Pesch, S. Shebs et al., *Debugging with GDB: The GNU Source-Level Debugger,* GNU Press, 2002. Online version: ninth edition, for GDB version 5.1.1, Jan. 2002, http://www.gnu.org/manual/gdb-5.1.1/, accessed 29 Dec. 2003.
56. J.G. Steiner, C. Neuman, J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", Proc. Winter 1988 USENIX Conf., pp.191-201.
57. M. Stytz, J. Whittaker, "Caution: This Product Contains Security Code", pp.86-88, *IEEE Security & Privacy* vol.1 no.5 (Sept./Oct. 2003), IEEE Computer Society.
58. Symantec, "What is the difference between viruses, worms, and Trojans?", http://service1.symantec.com/SUPPORT/nav.nsf/pfdocs/1999041209131106, October 1 2003.
59. Technologismiki (Athens, Greece), *Hackman* Hex Editor 7.05, *Hackman* Disassember 8.01, *Hackman* Debugger, http://www.technologismiki.com/en/index-h.html, accessed 29 Dec. 2003.
60. P.C. van Oorschot, "Revisiting Software Protection", pp.1–13, Proc. of 6th International Information Security Conference (ISC 2003), Bristol, UK, October 2003, Springer-Verlag LNCS 2851 (2003).
61. N. van Someren, A. Shamir, "Playing Hide and Seek with Keys", pp. 118–124, Financial Cryptography'99, Springer-Verlag LNCS 1648 (1999).
62. J. Viega, G. McGraw, *Building Secure Software*, Addison Wesley, 2001.
63. J. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, R. Vinciguerra, "An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java", Proc. 10th IEEE WCRE (Working Conference on Reverse Engineering), Victoria, Canada, Nov.2003.
64. VMWare Inc., VMWare Workstation (virtual machine), http://www.vmware.com, site accessed 29 December 2003.
65. N. Weaver, V. Paxson, S. Staniford, R. Cunningham, "A Taxonomy of Computer Worms", pp.11–18 in [51].
66. J. Wilander, M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", pp.149-162, Proc. of NDSS'03 (Internet Society): Network and Distributed System Security Symp., Feb. 2003, San Diego.
67. Winternals Software (Austin, Texas), *FileMon* monitoring tool, site accessed 29 December 2003, http://www.winternals.com.