AN IMPROVED MODEL FOR SECURE CRYPTOGRAPHIC INTEGRITY VERIFICATION OF LOCAL CODE

Christian Payne
School of Information Technology, Murdoch University, Perth, Western Australia

Keywords: Code integrity verification, malicious code, one-way hash functions.

Abstract: Trusted fingerprinting is a new model for cryptographic integrity verification of executables and related objects to protect users against illicit modifications to system programs and attacks by malicious code. In addition to a number of other novel features, trusted fingerprinting improves upon previous designs by managing the privileges assigned to processes based upon their verification status. It also provides greater flexibility as, in addition to globally verified programs, each user can independently flag for verification software relevant to their individual security requirements. Trusted fingerprinting also allows for automatic updates to fingerprints of objects where these modifications are made by trusted code.

1 INTRODUCTION

Numerous vulnerabilities in software (Dowd et al., 2007; US-CERT, 2006) and a proliferation of malicious code (Gordon et al., 2006; Kalafut et al., 2006) mean that compromise of computer systems is becoming increasingly difficult to prevent. One approach to mitigating these risks is the use of cryptographic techniques such as one-way hash functions or digital signatures to support verification of code integrity so that these intrusions may be detected and subsequently repelled. While code signing is used extensively for remote and distributed code (Gong et al., 1997; Microsoft Corporation, 2006), local code verification is far less common. Existing schemes have both practical and architectural limitations and have failed to recognize that code verification is only one piece of the puzzle. While verification can detect naive modifications, there are a number of ways that attackers can bypass these security checks. Preventing this requires that the privileges assigned to a program be determined based upon its verification status.

This paper presents a new model for integrity verification of local system code. Like some previous approaches it uses one-way hash functions to generate hash values or ‘fingerprints’ for security-related system objects. However, it improves upon previous designs by also considering the files a program interacts with at runtime and whether the program’s integrity has been verified or not. Based upon this a trust level is assigned to the process and its privileges are controlled by integrating the code verification model with a secure access control scheme. This allows trusted, verified code to be granted higher privileges than programs that have not been authenticated and therefore defends against more sophisticated redirection attacks. It also resolves practical issues such as verification of non-native executables (such as scripts) and allows the automatic updating of fingerprints belonging to data objects where these modifications are made by trusted code.

2 PREVIOUS APPROACHES

Use of cryptography to verify code integrity has long been identified as a means for detecting substitution or modification of system programs. The Tripwire software generates one-way hashes of important system files and is then run on a regular basis to recalculate these hashes and detect if these files have changed (Kim and Spafford, 1994a; Kim and Spafford, 1994b). However, Tripwire is a passive detection mechanism and its high detection latency means that there is typically a considerable window of vulnerability between a compromise occurring and the administrator being alerted to this event.

Consequently mechanisms which actively prevent the execution of compromised code have since been developed. CryptoMark involves embedding signatures in the ELF binary header with the kernel verifying this signature and potentially denying execution

Payne C. (2007).
AN IMPROVED MODEL FOR SECURE CRYPTOGRAPHIC INTEGRITY VERIFICATION OF LOCAL CODE.
In *Proceedings of the Second International Conference on Security and Cryptography*, pages 80-84
DOI: 10.5220/0002124300800084
Copyright © SciTePress

if this fails (Beattie et al., 2000). The I^3FS scheme uses a stacked filesystem that incorporates the storage of hash values for data objects with verification occurring upon access (Patil et al., 2004). Yet another approach utilises the code verification features of the TPM chip found in TCG-compliant computer systems to cache known-good 'measurements' of system objects and verify these upon access (Sailer et al., 2004).

Each of these designs suffers from one or more serious limitations. For example, the TCG-based integrity verification scheme has no convenient way for users to update hash values when legitimate changes are made to the corresponding file object. This is problematic from a security perspective as it means objects that may change frequently are likely to be omitted from verification. This creates scope for a verified program to be subverted by malicious modifications to the data file inputs that control its behaviour such as configuration files.

A similar problem affects CryptoMark as its dependence upon storing signatures inside the ELF binary executables themselves does not allow for verification of data files. This also precludes verification of non-natively executable scripts. Its designers suggest that a solution to this is the use of a separate table specifying signed objects and their signatures. However, the table then becomes the target of attack and leads to the more general problem of determining which system objects require verification and which do not. This needs to be reliable and immutable in the face of an attacker with superuser privileges otherwise the system can be tricked into mis-identifying a maliciously modified object as one that does not require verification. If this is not the case then an attacker can either insert new malicious code or redirect users to unwittingly execute code other than the intended trusted program. This problem is illustrated in the I^3FS design where a Unix inode value rather than a path is used to identify each file. This allows a privileged attacker to create a new program of the same name as one that is cryptographically verified but with a different inode number. A user executing this program will believe they are running a trusted, verified program but as the inode numbers will not match, the redirection will not be detected.

This issue can be resolved by requiring *all* objects (both code and data) to be verified but this is impractical (Reid and Caelli, 2005). Not only does it impose an unnecessary performance penalty but dealing with modified or newly created objects becomes problematic. An underlying cause of these limitations and weaknesses is that all programs execute with the same level of privilege, regardless of whether they have been verified or not. A novel solution is therefore to link the privilege with which a program executes with its verification status. This effectively limits the impact of insertion or redirection attacks by constraining the privileges of non-verified code. A new model for local code integrity verification that uses this technique will now be described.

3 OVERVIEW OF TRUSTED FINGERPRINTING

3.1 The Vaults Security Model

Vaults is an operating system security model that provides a key management infrastructure and a cryptographic locks and keys based access control scheme (Payne, 2003; Payne, 2004), in addition to the integrity verification mechanism described here. The access control model applies cryptographically-based read and write protection to selected files which are then accessed through tokens called 'tickets'. Users can also grant others access to their files by generating additional tickets for them. Note that these access controls apply in addition to existing discretionary access control lists or permissions. The model also facilitates the storage of a variety of items in secure repositories known as 'vaults'. Each user has their own vault which they are responsible for administering using specially designated software. However, there are also system vaults maintained by the kernel and security administrator. Of these, the one most relevant to trusted fingerprinting is the Global Public vault which can be read by all users but only modified by the security administrator.

The items stored in these various vaults include tickets and file encryption keys but users can also store secret values such as passwords that are associated with specific applications. These applications can then request access to the keys that are bound to them in a manner that is transparent to the user. Each user's vault is stored in encrypted form on the disk and loaded into memory when the corresponding decryption key is supplied. Access to this vault is restricted to trusted programs executed subsequent to vault decryption.

Trusted fingerprinting has been designed to integrate with the Vaults model and leverage the features it provides. For example, the fingerprints generated from system objects can be stored securely in the appropriate vault. As access to a vault is cryptographically controlled, this ensures the integrity of the fingerprint values. Fingerprints can then be verified when the corresponding program or file is accessed.

Additionally, the cryptographic access control architecture provides a convenient means for managing process privilege and confinement. However, trusted fingerprinting also improves the security of the Vaults model since it allows applications to be authenticated before keys are released to them.

3.2 Local and Global Dependencies

The existence of both the Global Public vault and individual user vaults gives trusted fingerprinting a unique two-tier design. The security administrator can specify programs with system-wide security significance to have their fingerprints verified upon execution by all system users. These fingerprints are stored in the Global Public vault and are readable by all. However, in addition to this each user can also store fingerprints in their own vault for programs relevant to them and their specific security requirements. This provides superior flexibility compared with previous approaches and recognizes that security requirements differ from user to user. Furthermore, it reduces the workload of the administrator as individual users can flag programs for verification without requiring administrative intervention.

Trusted fingerprinting also recognizes both that not all system objects require verification and that dependency relationships exist between those that do. For example, a program's secure behaviour at runtime may depend upon shared libraries, data files and even other programs. These dependency relationships are modelled using interconnected digraph structures that reflect the shared dependencies of different programs. This enables efficient and secure runtime verification of objects.

3.3 Process Trust Levels

Programs are assigned a trust level which is stored as part of their fingerprint record and determines the level of privilege that the process will normally have when executed. Four hierarchical levels of trust are defined and these are designated Level 0 to 3 (L0–3), with higher numbers indicating greater privileges. Unverified code is classified as L0 and cannot access a user's vault. The effect of this is that L0 processes cannot utilise any access tickets that the user possesses and are therefore unable to access cryptographically protected files where a ticket is required. Forcing all unverified programs to be unprivileged limits the damage from malicious code in the event of a successful redirection or insertion attack. This resolves a significant limitation of previous schemes.

Beyond this, verified code can still run with limited privilege by being designated L1 which results in the program being confined to a limited set of specifically identified file access tickets. This allows programs to execute with the minimum amount of privilege required and is particularly applicable to programs such as web browsers and e-mail clients that are exposed to large amounts of untrusted code and data. Applications labelled as L2 are considered to be fully trusted and have access to all of the user's tickets while the L3 designation is reserved for the software used to administer vault contents.

3.3.1 Determining Trust Levels

The trust level of a process at runtime depends primarily upon the value assigned by the user and this is the highest trust level it may hold. However, its trust level can vary depending on other factors. Specifically the trust level of a new child process is instantiated as the minimum of that program's user-specified trust level and its parent's current trust level[1]. Without this the secure operation of a trusted program could be subverted if it were executed by malicious code and supplied with dangerous parameters or inputs. This therefore prevents a malicious, untrusted program from misusing a trusted one. As a result it is not necessary to analyse the behaviour of each individual program in response to all possible inputs when assigning trust levels.

The manner in which the trust level of a program is decided is designed to reflect both applicable security semantics and the practical requirements of program operation. For example, L0 applications are unable to access cryptographically protected files where this requires a ticket. However, where discretionary controls allow, they may still access files that are only protected with the opposite mode to that requested; e.g., where read access is requested to a file that is only write protected. This effectively minimises the privileges of an L0 program by isolating it from any sensitive data while not limiting its privileges to such a degree that the user is forced to elevate its trust level in order for the program to operate correctly. Therefore, users need only increase an L0 program's trust level if there is a set of specific cryptographically protected objects that they wish the program to be able to access. Furthermore, even in this scenario, unless the program requires general access to cryptographically protected files, the user can set the trust level

[1]The only exception to this rule is the special case where an L3 program is executed and these can only be spawned from L2 processes. Because of this, L3 programs must be designed to prevent any subversion of their behaviour by parameters or other external inputs.

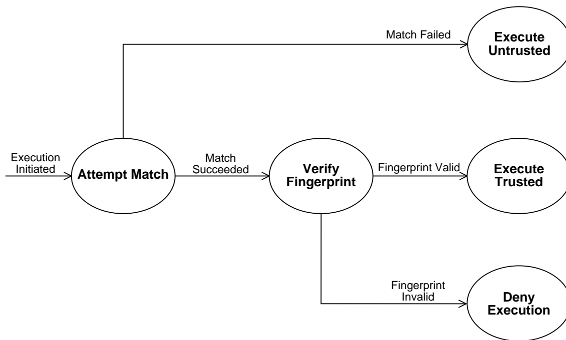

Figure 1: Overview of process execution, matching and verification.

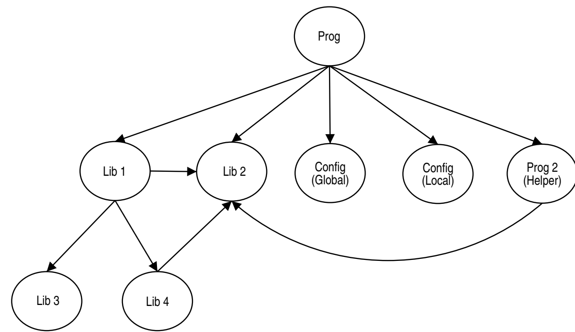

Figure 2: Example runtime digraph.

to L1, thereby confining the program to these specifically selected objects.

3.4 Matching, Verification and Execution of Programs

An overview of the process that occurs when a program is executed is given in Figure 1. When a program's execution is initiated, a matching process occurs where the user's vault and Global Public vault are searched for a fingerprint entry corresponding to the path of the file being executed. If no match is found then the program will execute as untrusted (L0), otherwise, if verification is successful, the trust level is set based upon the value in the matched fingerprint record and according to the rules described in the previous section. This ensures that the default behaviour is to minimise the privileges assigned to a program and this significantly mitigates the risks due to insertion or redirection attacks.

The local and global dependency digraphs identified through the matching process are then merged to construct a 'runtime digraph' for that program. This is maintained for the lifetime of the process and is used to identify and verify the relevant objects when accessed. An abstracted example of a hypothetical runtime digraph is given in Figure 2. In this example, the program is linked with two different shared libraries, which in turn have their own dependency arrangements as shown by the direction of the arrows. The program has both global and local (per-user) configuration files, both of which are identified in the runtime digraph as a result of the merging process. The program also has a helper application that it executes at runtime.

Once matching is complete and the runtime digraph constructed, the program binary and the shared libraries it depends upon are verified by calculating one-way hashes and comparing these against the fingerprint values in the runtime digraph. The program and shared library images are cached in memory between verification and execution. This excludes race condition-based attacks involving a privileged attacker modifying a trusted object in the time between verification and use.

3.5 Data File Dependencies and Automatic Updating

Non-executable data objects accessed by a trusted program at runtime are checked against the runtime digraph. If these are specified as a dependency for that program then these are also loaded into memory and verified. This allows for the files used by an application which impact on security to be specifically marked for verification without the performance impact of requiring all accessed files to be verified. This design also facilitates verification of non-natively executable programs such as scripts by specifying the script source code as a dependency of the natively executable interpreter or runtime environment. Trust levels can also be assigned to individual scripts and the effective trust level of the interpreter process is modified to be the minimum of its current level and that assigned to the data file dependency. This enables interpreted scripts and other non-natively executable programs to have different and independent trust levels to that of their interpreter.

As is the case with programs and libraries, data file dependencies are also cached at runtime. Because of this, modifications to these data objects by a trusted process are captured by the system and used to automatically update the user's fingerprint value for that object. This is a convenient and secure approach to dealing with modifications to trusted data that does not involve the level of administrative overhead required in previous file integrity verification

schemes. Automatic fingerprint updating can only be performed by a program which is a parent of the data file dependency in the program's runtime digraph. In this way the runtime digraph clearly defines the trust relationships between a program and the libraries and data files it depends on. However, users can also manually update fingerprint values in their vault using a specifically designated L3 application.

4 CONCLUSION

Trusted fingerprinting is a new model for local code verification which leverages the cryptographic infrastructure provided by the Vaults architecture. It resolves the weaknesses of previous approaches in that it actively prevents execution of maliciously modified objects, supports the verification of non-natively executable scripts, allows the automatic updating of fingerprints when modifications are made by trusted code and limits the privileges assigned to unverified processes. Other unique features include the use of digraphs to represent the relationships between programs and the objects they depend upon for their security and a two-tier approach where users can independently specify programs to be verified consistent with their individual security requirements in addition to global defaults. This recognizes the need for greater flexibility in local code verification architectures in order to avoid administrative overheads that otherwise represent a significant impediment to their adoption. Therefore, the adoption of the trusted fingerprinting code verification technique, in conjunction with the other features of the Vaults model, has the potential to significantly mitigate the effects of widespread malicious code and intrusion.

REFERENCES

Beattie, S. M., Black, A. P., Cowan, C., Pu, C., and Yang, L. P. (2000). CryptoMark: Locking the stable door ahead of the Trojan horse. White paper, WireX Communications Inc.

Dowd, M., McDonald, J., and Schuh, J. (2007). *The Art of Software Security Assessment*. Addison-Wesley.

Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R. (1997). Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

Gordon, L. A., Loeb, M. P., Lucyshyn, W., and Richardson, R. (2006). Eleventh annual CSI/FBI computer crime and security survey. Technical report, Computer Security Institute (CSI). http://GoCSI.com.

Kalafut, A., Acharya, A., and Gupta, M. (2006). A study of malware in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM on Internet Measurement*. ACM Press.

Kim, G. H. and Spafford, E. H. (1994a). The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computers and Communication Security*.

Kim, G. H. and Spafford, E. H. (1994b). Experiences with Tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-93-071, COAST Laboratory, Purdue University, West Lafayette, IN 47907-1398.

Microsoft Corporation (2006). Introduction to code signing. Online: http://msdn.microsoft.com/workshop/security/authcode/intro_authenticode.asp.

Patil, S., Kashyap, A., Sivathanu, G., and Zadok, E. (2004). I^3FS: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*.

Payne, C. (2003). Cryptographic protection for operating systems. Research Working Paper Series IT/03/03, School of Information Technology, Murdoch University, Perth, Western Australia.

Payne, C. (2004). Enhanced security models for operating systems: A cryptographic approach. In *Proceedings of the 28th Annual International Computer Software and Applications Conference: COMPSAC 2004*, pages 230–235. IEEE Computer Society.

Reid, J. F. and Caelli, W. J. (2005). DRM, trusted computing and operating system architecture. In *Proceedings of the 2005 Australasian Workshop on Grid Computing and e-research*, volume 44, pages 127–136.

Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. (2004). Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238.

US-CERT (2006). Quarterly trends and analysis report, volume 1, issue 2. Technical report, United States Computer Emergency Readiness Team. http://www.us-cert.gov.