

Trusted Objects

Philip L. Campbell, Lyndon G. Pierson, Edward L. Witzke
Sandia National Laboratories
Albuquerque, New Mexico 87175
{plcampb, lgpiers, elwitzk}@sandia.gov¹

Abstract: In the world of computers a trusted object is a collection of possibly-sensitive data and programs that can be allowed to reside and execute on a computer, even on an adversary's machine. Beyond the scope of one computer we believe that network-based agents in high-consequence and highly reliable applications will depend on this approach, and that the basis for such objects is what we call "faithful execution."

1 Background

This paper is an outgrowth of a question we raised amongst ourselves: what is the digital equivalent of the standard paper envelope? Envelopes provides authenticity—especially if there is handwriting both on the envelope and on the letter inside—and a good measure of confidentiality as well. Although envelopes are not difficult to tear open, it is difficult to conceal the evidence of tampering. They are unlike the 1,000 lb. steel safes we send through the mail in the digital world in the form of encrypted messages.

What we want is a digital message that, say, recorded who had read the message as it made its way from originator to destination, or at least recorded who the readers purported to be. Perhaps the digital message could go a little further, say, by authenticating those attempting to read the message. It could actively guard the message by demanding that it be allowed communication with the originator before giving read permission, for example. Further, the message could reveal only part of its contents, depending on who was reading it. Or perhaps it might do simple duties, such as assist the reader by displaying the data in various ways, or provide translations for different terminals or printers.

However, we noted that as soon as our digital message was able to do anything, such as authenticate or even just record names, it could do a number of things. Our active messages, then, are less like standard paper envelopes and more like genies with uncertain tempers.

This paper provides an implementation of these active, digital messages which we dubbed "trusted objects," their problems, their application, and their future.

2 Introduction

At the most rudimentary level, an "object" in software parlance is an association between particular data and particular programs. The two are combined in such a way that they can be referred to and manipulated as one entity. At this level an object assists a user by identifying and including within itself the programs that are intended to be run on the accompanying data. Unlike data or the written word, an object opens a qualitatively new world since it is able to interpret itself [11].

1. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL8500.

At the next level, objects can protect the data by providing “encapsulation.” Data that is encapsulated cannot be accessed directly; it can only be accessed by invoking a program within the object itself. Encapsulation is usually referred to as a feature of objects, just as we have done in the previous sentences. However, it is an object’s environment, not the object itself, that gives the encapsulation teeth. If an outside program, such as a debugger, can circumvent the environment, then the encapsulation has no strength and is left toothless.

At the third level, objects include their environment. Because the environment is included in the object, a user cannot circumvent the encapsulation by breaching the environment. We call these objects “trusted” since the developer can with equanimity allow them to be executed on an adversary’s machine, even if the objects carry sensitive data.

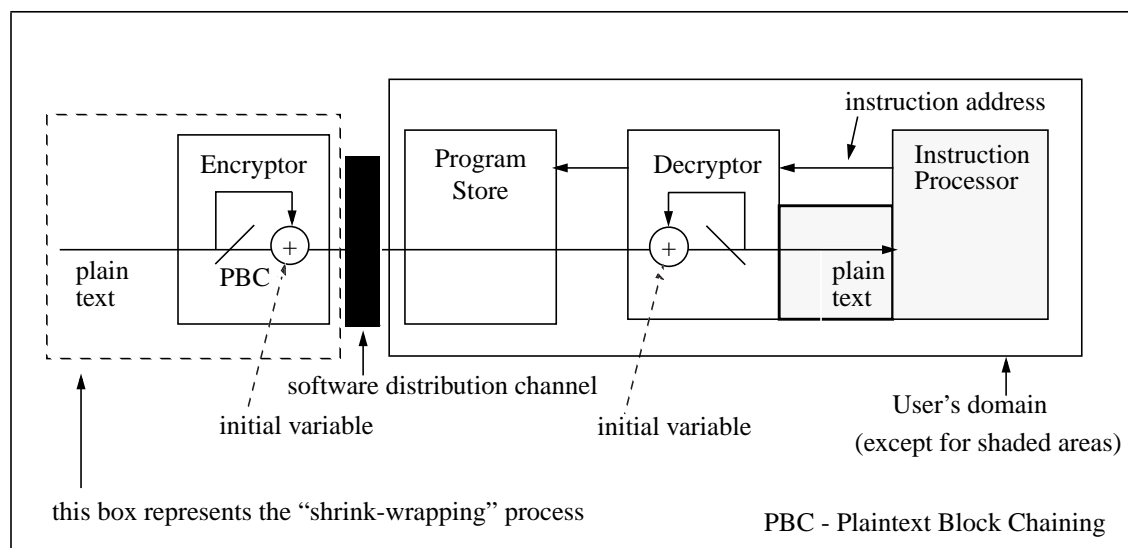
A “trusted object,” then, is an association of data and program that provides encapsulation and includes its own environment. Trusted objects are designed to be able to execute in an untrusted environment, such as on an adversary’s machine. The adversary has access to the binary image of the object as it resides on disk, but the adversary cannot read the data or decipher the program. The value of such objects is that they can carry their security with them, so to speak. They can authenticate users, for example. They can communicate with their originator, and if the user refuses to allow this, the object can erase itself. Such objects require what we call faithful execution. Software protection schemes, such as Albert & Morse’s ([1] and the Appendix) and Best’s [3] provide similar functionality.

3 Implementation: Faithful Execution

“Faithful execution” is the technique that protects executable files so that they can be trusted to run as designed even in an environment over which the designer has no control. The executable files are protected from substitution, corruption, and infection.

For a trusted object, faithful execution enables the object to execute without any of its pieces being accessible. The object is protected by encryption while it sits in memory; it is decrypted only when it is actually in the CPU, as it is needed. The fetch addresses from the CPU to memory need not be encrypted. Figure 1 presents the process diagrammatically.

Figure 1. Faithful Execution



Since Plaintext Block Chaining (PBC) [8] mode is used, the adversary is unable to use one of the encrypted instructions sitting before him in memory to mount a block replay. Since he does not know the cryptographic state required for decryption he is unable to construct a string of bits that represents a legal instruction. In addition, since PBC mode is not self-synchronizing, then an attack will upset the synchronization, and the decrypted instructions will likely never return to synchronization.

Schemes to protect software from piracy may provide similar functionality. For example, Albert & Morse's approach can be thought of as an instance of faithful execution. Best's "Cryptographic Micro Processor" is another example.

4 Problems

Trusted objects have two problems that must be overcome before they can become widely used. The first problem is the protection of the encryption keys. The second problem is that they put the receiver at risk.

4.1 Key Management

Encryption takes a big secret, so to speak, and turns it into a small secret. Instead of having to protect the entire message, we only have to protect the key. This is easier for at least two reasons. The first is that the message and its protection are separated, enabling concentration of protection on what needs it most and making management of that protection easier. The second reason is that the key is smaller, and for this reason alone we presume it is easier to manage.

To secure an insecure channel, encryption provides good service. The model is that of a tube, open at either end. The message on either side of the tube (or channel) is unencrypted, and, in principle, does not need to be encrypted except when it is in the channel, which we presume is for a relatively short amount of time. The key is thus not bound to the message, except for that short period. If the message needs to be put through the channel again, the key that is used need have no relation to the previous key.

However, if the message is a trusted object, then the tube model no longer fits. One end of the tube is closed off. But even that is not quite correct. Better, but more discursive, is the model of a traveling businessman who gives gifts to those who know the password, but never returns home and does not trust any channel to enable a changing of the password. Managing the keys for trusted objects is a serious problem.

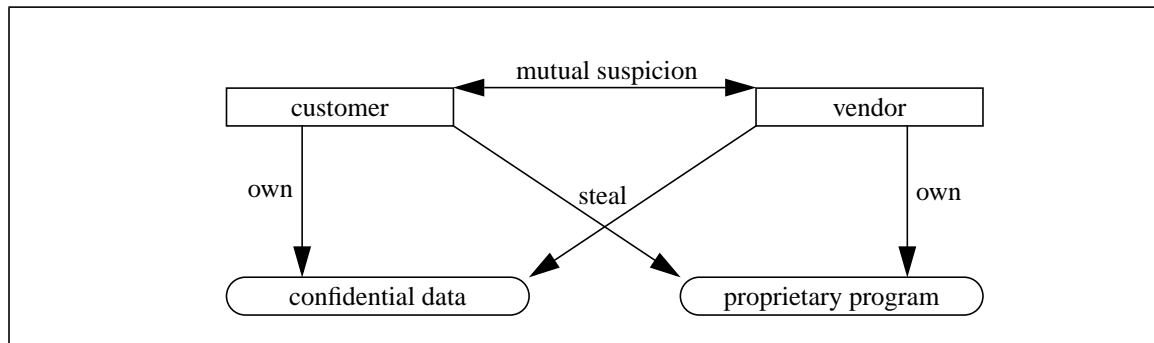
4.2 The Problem of Mutual Suspicion

Another problem for trusted objects is that they put the receiver at risk. In order to get any benefit from a trusted object, the receiver must execute it. At that point the object has control and can wreck havoc, if the object is malicious. In general, no receiver can identify a malicious trusted object without first executing it.

The problem that the sender and receiver of a trusted object face is the same that a seller and buyer face: mutual suspicion. Software vendors and customers have this relationship since the vendor is suspicious that the customer will steal the code by copying it and thus rob the vendor of purchase fees, and the

customer is suspicious that the vendor will steal the data by copying it and thus rob the customer of the information. The relationship is depicted in Figure 2.

Figure 2. Mutual Suspicion (from [5])



The customer could also be suspicious that the vendor's code will damage the customer's data or code or machinery. For example, the code could change a memory bit at random each time it executes. This would be difficult to detect, but it is not difficult to imagine the problems that this could cause.

The solution that the software marketplace uses today is customer-centric, putting the vendor at risk. Vendors try to protect themselves by using some form of copy protection. This is generally a nuisance to the customer, so vendors apply as little of this as possible, and the losses due to software piracy are estimated to be 27% in the United States, at a value of \$2.8 billion [6]. Vendors also protect themselves by indirect tactics, such as frequently generating new versions. The new versions help create a revenue stream; they also help keep competitors at bay; but they also help reduce piracy, since these new versions are cheap for those who have a valid registration number. Meanwhile, other pieces of software are also marching through upgrades, so the pirate must keep robbing in order to keep abreast, or at least to avoid incompatibilities. The vendors, as a group, hope that at some point it is cheaper just to buy the software.

An alternative approach could be called vendor-centric: the customer sends the data to the vendor. In this approach the vendor will be better able to protect the code. But now the customer has to worry about the safety of the data. The customer could first encrypt the data and ask the vendor to compute on that, but the vendor might be unwilling as a matter of policy to send encrypted data back to the customer, since this practice could invite a security leak.

A third approach is to introduce a third-party, preferably one that is well known and is assumed to be an enduring member of the market, with a future revenue stream that would be threatened by a lapse in reputation. IBM, for example, provides this service via its "cryptolope" technology ([4], [9], [10]). Cryptolopes are cryptographic envelopes. They are encrypted with a clear-text preamble. A vendor engages IBM to sell the vendor's digital wares by distributing them within these cryptolopes. When a receiver decides to buy, after reading the preamble, then the receiver contacts IBM to get a decryption key, at which point money changes hands. Control over the data within the cryptolope is not necessarily entirely lost at this point: the key could unlock only a viewer within the cryptolope, for example. Note that we are almost back to the same problem of the receiver having to give control to code that the receiver had no hand in producing. However, the presence of the trusted-third party to whom the receiver has made a payment for services is an avenue for recourse for the receiver in case the cryptolope damages the receiver's machine. IBM is not the only vendor to provide this type of service. Note that cryptolopes and trusted objects are not the same, even though both can contain both code and data: the former requires contact with a third-party for each transaction and executes clear-text code.

The commercial world cannot solve the problem of mutual suspicion once and for all since unscrupulous characters will probably always be with us, but the quantity of trade worldwide indicates who is currently winning the battle. In that commercial world there is a meeting ground, called a marketplace, where both parties extend their part of the bargain; if either claims to have been swindled, there is a means of recourse. We do not yet have such a marketplace for bits.

5 Applications

We believe that network-based agents, for example, in high-consequence and highly reliable applications will depend on this approach. How else could the results of the agents be considered to be of value? We also see a place for trusted objects in secure communications, which we believe will increasingly depend on the application interacting with the user on the user's platform in a secure fashion.

Another use is in treaty verification. In this application a program transmitting weapon status information or sensor data must be trusted both by the country in which the program is executing and by the country owning the remote device or sensor. Both countries must trust the executing program to do exactly what it purports to do.

In general, the value of trusted objects is that they may allow us to reverse the fortress mentality of traditional computer security in which the good guys are surrounded by the bad guys, and replace it with what we might call a spies-everywhere mentality in which the good guys surround the bad guys.

6 Future

Computing is so new that security has always been a second-thought, if it is considered at all. We have been preoccupied in getting computing systems to work and keeping up with the pace of innovation, and security has been given short shrift. But as our world becomes increasingly dependent on computing—particularly networked computing—security will be given attention. One path to security is to physically control the computing equipment—all of it. This is possible, but stifling. Encryption allows us to relinquish control of the channels without relinquishing security. Trusted objects may allow us to relinquish control of almost everything else.

7 Conclusions

Although we have not yet found the digital equivalent of the standard paper envelope, we have explored territory that provides new possibilities for security in the expanding, increasingly network-centric world of computing, namely what we call trusted objects. These objects provide security by taking their security with them, so to speak. We believe that these objects provide the kind of security that will be of increasing interest in the future.

Acknowledgment

We would like to thank Anne Van Arsdall for her review of this paper.

References

- [1] D. J. Albert, S.P. Morse, "Combatting Software Piracy by Encryption and Key Management." IEEE Computer, April 1984, pp. 68-73.
- [2] Mikhail J. Atallah, Konstantinos N. Pantazopoulos, Eugene H. Spafford, "Secure Outsourcing of Some Computations." COAST Laboratory. Department of Computer Sciences, Purdue University. West Lafayette, IN.
- [3] R. Best, "Preventing Software Piracy with Crypto-Microprocessors." Proc. Comcon, Spring 1980, IEEE-CS Press, Los Alamitos, CA, pp. 466-469.
- [4] "Cryptolope Container Technology." A White Paper by IBM Internet Information Technologies. April 1997. 22 pages.
- [5] D. E. R. Denning, *Cryptography and Data Security*. Addison-Wesley Publishing Co., Reading, Mass., 1982.
- [6] Dorothy E. Denning, "Information Warfare and Security." Addison-Wesley, Reading, MA. 1999.
- [7] J. Feigenbaum, "Encrypting problem instances or ..., can you take advantage of someone without having to trust him?" Advances in Cryptology-CRYPTO '85, pp. 477-488.
- [8] Jansen, Cees J. A., "Investigations on Nonlinear Stream Cipher Systems: Construction and Evaluation Methods." Philips usfa B.V., the Netherlands, 1989.
- [9] Pete Loshin, "IBM's Digital Shrinkwrapper." BYTE, August 1997, p. 138.
- [10] Matt McKenzie, "Copyright Protection: Understanding Your Options." Seybold Report on Internet Publishing, vol. 1, no. 4, p. 6-9.
- [11] M. Mitchell Waldrop, "Is there an information revolution?" Chapter 1 of "The information revolution and international security." edited by Ryan Henry and C. Edward Peartree. 1998. The Center for Strategic and International Studies.

Appendix: Albert & Morse's System

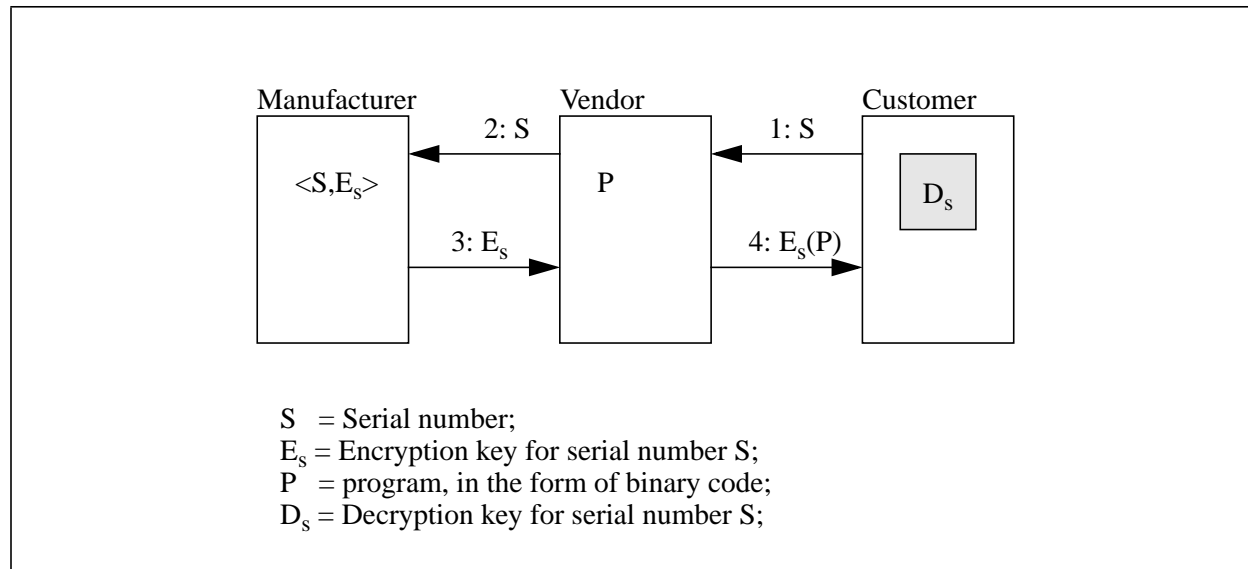
Albert & Morse [1] propose a solution to software piracy that requires the chip maker to provide the chip with a decryption key and a serial number. The former is not visible to the customer; the latter is. The protocol for safe exchange of software is as follows:

1. The software customer requests a copy of the software from the vendor by sending the serial number of the customer's CPU to the vendor.
2. The vendor sends the serial number to the manufacturer.
3. The manufacturer sends back to the vendor the encryption key corresponding to the serial number.
4. The vendor sends to the customer the program encrypted with the encryption key for the customer's CPU.

5. The customer instructs his CPU to enter decryption mode, then he runs the encrypted program received from the vendor.

This protocol is shown diagrammatically in Figure 3.

Figure 3. Sharing Software



The purpose of this protocol is to protect the software from piracy. The customer can view the encrypted copy of the software residing on his disk. He can make any number of copies of it, which he will probably want to do in order to provide backup capability in case of a disk crash or an inadvertent deletion. None of these copies will run successfully on any machine that does not have the appropriate decryption key. So the customer can give a copy of his software to someone else, but that other person will be unable to run the software successfully on their machine. The result is that this avenue of piracy is closed: if that other person wants to execute the program, they have to go through the same route as our customer, which involves paying a fee.

A second avenue for piracy is for the customer to view the software on the disk and to record the instruction sequence. With this sequence in hand, the customer can create their own copy of the software. This is a tedious, error-prone process, so much so that the customer has to be highly motivated to complete the task. Nevertheless, because the software on the disk is encrypted, the customer is unable to determine the instruction sequence and thus unable to steal the software via this avenue.

One of the problems with this protocol is that the encryption key is allowed to leave the manufacturer's control when a copy of it is sent to the vendor. The customer could set himself up as a vendor, get the key, then break the protocol and get a cleartext copy of the software. This problem could be overcome by prohibiting the manufacturer from releasing a copy of the encryption key, a solution requiring the vendor to send the program to the manufacturer. However this requires that the vendor trust the manufacturer. There is also the problem that the customer could masquerade as a manufacturer—just new on the market!—and get the program in one step. Alternatively, the vendor could encrypt the program with a key of his own choosing before sending it to the manufacturer, who would encrypt it with the customer's key and return the result to the vendor, who would decrypt the first encryption, leaving the second encryption in place. This is referred to as computing with encrypted data, and it is not clear how secure it is, though progress is being made ([7], [2]).