Defense-Enabled Applications

Franklin Webber, Partha P. Pal, Richard E. Schantz and Joseph P. Loyall
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
{fwebber, ppal, rschantz, jloyall} @bbn.com

Abstract

Some applications can be given increased resistance to malicious attack even though the environment in which they run is untrustworthy. We call any such application "defense-enabled". This paper explains the principles behind defense enabling and the assumptions on which it depends.

1 Introduction

Ideally, one defends a computer system against malicious attack by identifying, in a security policy, what one wants to protect and then by implementing that protection in hardware and software. The implementation is called a trusted computing base (TCB) [12]. The TCB is trusted not to violate the security policy itself[1], and, in most systems, it is also trusted to prevent other, possibly malicious, software from violating the policy. In a distributed system, the TCB usually includes key parts of the operating systems that run on network hosts and of the network communication paths between these hosts.

In practice, many computer systems today have no such trusted computing base. Many others have a design for a TCB but its implementation is seriously flawed. There are several reasons for this situation:

- It is hard to keep the TCB for a complicated system simple enough to warrant trust.
- It is hard to modify the TCB while maintaining trust, because even simple changes to the TCB can have unforeseen effects that undermine its protection.
- It is hard to redesign an existing system to create a TCB if none was planned for originally.

[1] Technically speaking, the TCB is trusted to violate the security policy so long as the effect of that violation is not equivalent to allowing non-TCB software to violate the policy.

In fact, many of the world's computer systems today run operating systems and networking software that are far from the TCB ideal. These systems may lack any security policy, can be damaged using well-known attacks, and therefore cannot be trusted to protect anything. These systems will continue to be used because of the many applications that depend on them, but are unlikely to be redesigned to be more trustworthy.

Given this situation, one might ask: "What kind of defense is possible for systems that can be accessed by malicious attackers but lack trustworthy operating systems and networking to protect them?"

In principle, the answer is "None." A determined attacker can, with sufficient work, defeat whatever flawed protection is offered by the operating systems or networking, thus gaining privileges that can be used either to kill the system completely or to corrupt it some other way. Although one might try to protect data using encryption and digital signatures that are computationally infeasible to break [10], when that data is processed by the system it will almost certainly become vulnerable to an attacker with enough privilege. Note that encrypted data is worthless unless it is decrypted at some time, and it can be read at that time by a privileged attacker; also note that digitally signed data must be re-signed when it is modified, and an attacker who gains the privilege to re-sign data can forge new, corrupt data as well.

In practice, though, an attacker may not have the skill, perseverance, preparation, or time needed to carry out the attacks that are possible in the worst case. Some attackers rely on prepackaged attack "scripts" and do not have the skill to repair the scripts if they fail. An attacker who meets unexpected obstacles may look elsewhere for easier targets rather than persevere in an attack. An attacker who is not prepared in advance to circumvent the protection in a specific system will be more likely to trigger intrusion detection alarms [4]. In any case, the more time an attacker takes, the more vulnerable he is to being detected and stopped by system administrators.

In summary, system protection is not perfect, but attacks and attackers aren't either.

This paper makes a distinction between *protection*, which seeks to prevent the attacker from gaining privileges, and *defense*, which includes protection but also seeks to frustrate an attacker in case protection fails and the attacker gains some privileges anyway. Protection mechanisms are static and proactive; defense mechanisms enhance the protection mechanisms with a dynamic strategy for reacting to a partially successful attack. Both protection and defense aim to keep a system functioning, but protection tends to be all-or-nothing, either it works or it doesn't, whereas defense can choose from among a range of responses, some more appropriate and cost-effective than others.

2 Defending Critical Applications

The goal of defense is the correct functioning of one or more *critical* applications. These applications are critical in the sense that the functions they implement are the main purpose of the computer system on which they run. Defending other applications in the same environment is not a primary goal. Neither is defending the application's environment itself, e.g., the operating systems and networks that support the critical applications. Defending the environment is important only so far as it helps to defend the critical applications themselves.

We say that an application that does not function correctly is *corrupt*. A corrupt application might deliver bad service or it might fail to deliver service at all. The goal of defense, then, is to prevent or significantly delay corruption of critical applications.

An application can become corrupt due to various causes:

- either because of an accident, such as a hardware failure, or because of malice;
- either because flaws in its environment cause a loss of protection that allows it to be damaged or because flaws in its own implementation cause it to misbehave.

The main concern of this paper is corruption that results from a malicious attack exploiting flaws in an application's environment. We assume that this is by far the most likely cause of corruption and so the other causes will be neglected in this paper. This assumption is reasonable because:

- Malicious attacks, which are directed and intentional, are far more effective in corrupting an application than accidents, which happen randomly.
- Flaws in the application's implementation can be corrected more easily than flaws in the application's environment, and the latter are likely to be better known to attackers and exploited by them.

Note that we are assuming we can modify or extend the design and implementation of the critical applications. This is in sharp contrast with the design and implementation of the environment, which we assume is almost completely beyond our control. In other words, we must live with flaws in the environment but, because our goal is defending critical applications, we will expend the effort to make those applications much more trustworthy than the operating systems and networks on which they depend.

We say that an application is *defense-enabled* if mechanisms are in place to cause most attackers to take significantly longer to corrupt it than would be necessary without the mechanisms. In other words, an attacker must not only defeat protection mechanisms in the environment, he must spend additional time defeating defense mechanisms added to the application.

The central factor in both attack and defense is *privilege*. An attack succeeds when the attacker gains privileges that allow him to corrupt some critical application. Defense enabling, therefore, must succeed either by keeping the attacker from gaining such privileges or by keeping him from using those privileges effectively. Defense enabling, therefore, can be divided into two complementary goals:

1. The attacker's acquisition of privileges must be slowed down. How this can be done is the topic of section 3.
2. The defense must respond and adapt to the privileged attacker's abuse of resources. Mechanisms for doing this are the topic of section 4.

Both goals are important. The first one makes the protection in the application's environment last longer. The second one makes the attacker work harder to use newly-gained privileges to corrupt a critical application. Because we have assumed that a determined attacker cannot be delayed indefinitely, both goals are needed for defense.

Defense enabling is organized around the application to be defended rather than around the operating systems and networks that support it. This follows simply because the application can be modified whereas the environment, for the most part, cannot. Section 4 explains that many defense mechanisms will tend to be placed into middleware [1], which is not part of the environment (in the traditional sense we have defined it here) but is still separate from the application's functionality. This separation keeps the defense mechanisms from complicating each application's design and allows for easy reuse in multiple applications.

3 Slowing the Acquisition of Privilege

Defense enabling depends on slowing the spread of privilege to attackers. To see this, note that if privileges could be gotten instantly, the attacker could immediately grab all the privileges needed to stop all application processing and thus to deny all service. No defense would be possible against this unlimited attack.

To help slow the spread of privilege, we divide the system into several *security domains*, each with its own set of privileges. The intent is to force the attacker to take more time accumulating the privileges needed to corrupt the applications. This will be true if:

- Each critical application has parts that are intelligently distributed across many domains so that privilege in a set of several domains is needed to corrupt it. This distribution of parts will be discussed in section 4.

- The attacker cannot accumulate privileges concurrently in any such set of domains. This constraint will be discussed later in this section.

A security domain may be a network host, a LAN consisting of several hosts, a router, or some other structure. The domains are chosen and configured to make best use of the existing protection in the environment to limit the spread of privilege. The domains must not overlap; for example, if the domains are sets of hosts then each host is in exactly one domain.

Each security domain may offer many different kinds of privilege. The following hierarchy is a minimal set that is typical in many domains:

- **anonymous user privilege**: allows interaction with servers in a security domain only via network protocols such as HTTP that do not require the client to be identified;

- **domain user privilege**: allows access only to a well-defined set of data and processes in one particular security domain (e.g., the user must "log in" to get this access);

- **domain administrator privilege**: allows reading and writing of any data and starting and stopping any processing in one particular security domain (e.g., "root" privilege on Unix hosts).

This hierarchy is listed in order of increasing privilege. Each of these privileges subsumes all the previous ones.

To increase the protection of critical applications we create a new kind of privilege in each domain:

- **application-level privilege**: allows interaction with a defense-enabled application using application-level protocols (e.g. CORBA calls that query the application or issue commands).

An attacker with application-level privilege would find it easy to control, and thus corrupt, an application, so defense enabling must make it hard for an attacker to get this privilege.

Application-level privilege is a key part of defense enabling. It differs from other kinds of privilege in that (a) it is not part of the environment but is created specifically to defend an application (b) it uses cryptographic techniques (which will be described later) (c) it does not subsume any of the other kinds of privilege and it is not not subsumed by any of them. In particular, gaining domain administrator ("root") privilege does not guarantee application-level privilege; this will be explained shortly.

A malicious intruder will attack a critical application by collecting the privileges needed to damage its integrity or to stop it from providing service. Using the set of privileges just listed, there are three ways for an attacker to gain new privileges:

1. by converting domain or anonymous user privilege into domain administrator privilege (e.g., exploiting bugs in trusted services, such as `sendmail`, that have domain administrator privilege already);

2. by converting domain administrator privilege in one domain into domain administrator privilege in another (e.g., using "root" in one domain to log in as "root" in another);

3. by converting domain administrator privilege into application-level privilege (e.g., using "root" privilege to invoke unauthorized application commands).

The attacker must be slowed down or prevented from gaining new privileges in each of these ways. How to do this will depend on the nature of the domains and therefore no generally-applicable rules can be given. However, the common case at issue today is security domains that are sets of network hosts. The following discussion applies to that case.

First, the attacker will try to convert domain or anonymous user privilege into domain administrator privilege by exploiting operating system security flaws. As explained in section 1, we assume this will always be possible. We also assume that it takes some time, possibly only a matter of minutes, but it is not instantaneous. The time it takes can be maximized by configuration of hosts and firewalls, for example, by applying the latest operating system patches, disabling or blocking unnecessary network protocols, and making the password file unreadable.

Second, the attacker can be prevented by proper host configuration from converting administrator privilege in one domain into administrator privilege in another. For example, hosts in different domains must not respect each other's privileges. This forces the attacker to start from scratch when trying to gain privilege in each domain.

Once having become a domain administrator, the attacker can quickly damage application processes in that domain simply by stopping them. With this privilege, he can bypass the operating system access controls that would normally prevent this damage. This damage, though, is contained because the application is distributed across many security domains.

Third, a defense-enabled application must use cryptographic techniques to prevent the attacker from gaining application-level privilege. An attacker having this privilege can do more damage than a domain administrator because direct attacks on the application cannot be confined to a single security domain. With application-level privilege, the attacker masquerades as part of the application itself, bypassing its access controls and causing it to behave incorrectly by sending it bogus commands and data, which the application itself propagates across the boundaries between security domains. The following techniques are therefore an essential part of every defense-enabled critical application:

- no application process can be started without authentication, e.g., executables are stored on disk encrypted with passwords known only to authorized users and other application processes;
- all communication between application processes is digitally signed with private keys known only to the application itself and communication uses sequence numbers to prevent replay.

Using these techniques will make it hard for an attacker, even one with domain administrator privilege, to masquerade as part of the application. Assuming the encryption is unbreakable, the attacker will be unable to corrupt the application process' code on disk. Assuming the digital signatures are unbreakable, the attacker will be unable to disrupt communication.

In principle, a domain administrator can gain application-level privilege with enough effort. For example, the administrator can read the core image of a running process, modify it to change the process' behavior, or search it to find the private keys used for digital signatures. This attack could be made harder with techniques for concealing or randomizing the location of data, e.g., passwords, within a core image, but the attack would still be possible. To counter this attack directly, the application must be made to confine application-level privilege, most likely using "Byzantine" fault tolerance techniques [2]. In practice, though, the effort needed for this kind of attack is likely to be much greater than the effort needed simply to kill all application processes in the domain, followed by attacks on other domains.

Finally, the attacker must not be able to gather privileges in many domains concurrently. This constraint means that an attack on an application in many domains cannot go just as fast as an attack on one domain.

An attack that proceeds sequentially, rather than concurrently, is called a *staged* attack; defense enabling relies on the attacker using only staged attacks. We can either assume that staged attacks are necessary or try to make them so. As a practical matter, most attackers will gather privileges sequentially as they explore a system's infrastructure, so staging may be a reasonable assumption. On the other hand, some attacks can be automated and carried out many times in parallel, in which case staging must be enforced.

One way to force an attacker to use a staged attack is by restricting all network communication to be local, i.e., each host can communicate only with its nearest neighbors and any networking protocols that allow communication with more distant hosts are disabled. As an example of how to do this, assuming a network of hosts equipped with standard Internet protocols [11]:

- Use a network infrastructure consisting of LANs connected by hosts acting as routers.
- Disable IP forwarding in all host routing tables.

This prevents an attacker from directly gaining privilege on an arbitrary host: he must first gain domain administrator privilege on neighboring hosts, re-enable IP forwarding on them, then attack neighbors of those hosts, etc. Thus some steps in the attack are forced to be done in sequence, not in parallel.

Unfortunately, disabling IP forwarding will not only impede the attacker, it makes building a distributed application harder. So, one must additionally create the effect of IP forwarding within each application, rather than relying on an infrastructure that implements Internet protocols:

- Replicate each application service on each host used for routing and multicast every service request by repeating it to replicas on neighboring hosts.

Having the application do the forwarding of its own communication also means that that communication is protected from an attacker who does not have application-level privilege.

This section has shown how defense enabling makes an attacker take longer to collect privileges. The next section shows how this extra time can be used for defense.

4 Competing for Control of Resources

The traditional approach to computer security treats the attacker and defender asymmetrically: the defender has domain administrator privilege, the attacker does not. The defender is given that privilege initially and uses that privilege to set up static protection both for critical applications and to maintain the asymmetry, i.e., the attacker must never get domain administrator privilege for himself.

In contrast, defense enabling assumes the attacker will eventually gain domain administrator privilege in some security domains, and in those domains the attacker and defender will be in symmetrical positions. What then? Section 3 showed how the defender can set up a new kind of privilege at the application level and try to protect it using cryptography. But the defender can also use domain administrator privilege to contest the attacker's control of domains. This section discusses mechanisms to use in that competition for resources.

Defense enabling includes the following tasks:

- **replicating**: Creating multiple security domains is not by itself sufficient to force the attacker to spend more time collecting privileges: if some domain were a single point of failure for the application, the attacker would need only to gain domain administrator privilege in that domain and kill application processes there. Clearly the application must be distributed redundantly across the domains.

  The simplest solution is to replicate every essential part of the application and place the replicas in different domains. Doing this turns the problem of defense into a problem of fault tolerance, where a "fault" is the corruption of a single replica by the attacker. The replicas must be coordinated to ensure that, as a group, they will not be corrupted when the attacker succeeds in corrupting some of them. Many protocols exist for fault tolerant replica coordination [9].

  Note that by creating and enforcing application-level privilege we may be able to simplify the fault tolerance problem to be solved. If the attacker cannot gain application-level privilege then application replicas will, at worst, crash when corrupted, and so it will not be necessary for the application to use more expensive protocols that protect against "Byzantine" corruption [2]. On the other hand, if the attacker can gain application-level privilege, such protocols are needed.

- **monitoring**: Intrusion detection systems (IDSs) [4] will be a part of this task, to collect data at the infrastructure level about possible attacks. Data collected at the application's level is also desirable, though, because it can give a more comprehensive view of the nature of the attack and more insight into potential remedies, and because it is more relevant to the needs of the application. Two kinds of monitoring are important at the application level:

  1. quality-of-service (QoS): whether the application is getting the QoS it needs from its environment and whether it is providing the QoS required by its users. A decrease of either QoS measure is an indication of a possible attack.
  2. self-checking: whether the application continues to satisfy invariants specified by its developers. A violation of such invariants is an indication that the application may be corrupt, possibly because the attacker has gained application-level privilege.

- **counterattacking**: If the source of an attack can be diagnosed with high confidence, resources can be denied to the attacker, for example, by killing the attacker's processes, denying the attacker bandwidth, or blocking communication from hosts running corrupt processes.

- **adapting**: If the attacker denies resources to a critical application, for example by killing application processes or flooding communication channels, the application must try to adapt to restore the QoS it needs. There is a wide variety of possible adaptations. The next section describes a classification scheme for defensive adaptations and gives several examples.

4.1 Classifying Defensive Adaptation

An application's defense will use one or more kinds of adaptation to counter a particular attack. This section classifies, in several dimensions, a basic set of potential adaptations.

In one dimension, shown vertically in table 1, adaptations differ by the level of system architecture at which they work. At the highest level, an application can choose to change its own behavior in the face of an attack, either finding an alternate way to proceed or degrading its service expectations. At the next lower level, the application can use QoS management support to try to make its environment offer the QoS it needs. At the lowest level, the application uses services from the operating system and network level to counter the attack, for example by changing details of how application components communicate.

In another dimension, shown horizontally in table 1, adaptations differ by how aggressively the attack can be countered. At best, the attack can be defeated, i.e., the effect of the attack on the application can be completely canceled. Second best is for the application to work around the attack, avoiding its effects. Finally, if the attack can neither be defeated or avoided the application can make changes to protect against similar attacks in the future.

Although table 1 shows at least one kind of adaptation for each of the nine possible boxes, the set of adaptations is not intended to be comprehensive: undoubtedly others can be invented or would be available with specific operating systems. There may also be other useful categories; for example, the table does not show any "honeypot" defenses in

	Defeat Attack	Work Around Attack	Guard Against Future Attack
application level	retry failed request	redirect reqst; degrade srvc	increase self-checking
QoS mgmt level	reserve CPU, bandwidth	migrate replicas	tighten crypto, access control
infrastructure level	block IP sources	change ports, protocols	configure IDSs

Table 1. A classification of defensive adaptations

which an attacker is lured into wasting effort on a decoy. In spite of these caveats, though, this set of adaptation mechanisms seems to offer a useful variety of options for creating a strategy for responding to attacks.

A third dimension for classifying adaptations is according to the kinds of attack they work against. In table 1, essentially two broad kinds of attack are countered:

1. direct attacks against the application, for example by disrupting the communication between its parts;

2. indirect attacks, in which resources the application needs are denied.

Direct attacks are countered by the mechanisms working at the application level, plus the use of encryption. An indirect attack might be countered by any of the mechanisms in the table but generally lower-level mechanisms can be more focused. For example, configuring a firewall to block packets from a particular source is a highly focused defense, but one that needs detailed information about the attack to have been collected first. At the QoS level, flooding the network can be countered by bandwidth reservation, overconsumption of CPU by scheduling and priorities, crashing of a node running an application component by migrating the component elsewhere, and relatively privileged operations can be disabled using access control if there is a high risk that they might be used maliciously.

A fourth dimension for classifying defenses is whether a mechanism can be used for protection from attack as well as for response to attack, or just for response alone. Mechanisms in the table's right-hand column, plus CPU and bandwidth reservation, can be used for protection. Why not always turn these strategies "on" for best protection? Because some of these defenses, e.g., an IDS configured to be very sensitive to attacks, have significant costs and so need to be used sparingly, and others, such as disabling highly privileged operations, impede the normal functioning of the system and so should be used only when necessary.

Incorporating many or all of these adaptation mechanisms into a single application can greatly complicate the application's design. Fortunately, every one of these mechanisms is orthogonal to an application's functionality, i.e., the application should compute the same results regardless of whether or how many defense adaptations have been used. In other words, every one of these adaptations changes *how* an application computes its results, not *what* results are computed. This orthogonality allows the design of defenses to be separated from the design of functionality.

It is natural to separate the design of functionality from the design of defenses by putting the latter into middleware [1]. The functionality can be designed first, then a strategy for defensive adaptation added later. Ideally, the defensive strategy and the mechanisms it uses would be reusable in many different applications, but this is not always possible. For example, access controls are specific to an application, and self-checking of application invariants will depend on application-specific data structures. These mechanisms seem to be exceptions, though: most of the other mechanisms in table 1 are reusable across applications.

5 Related Work

We are implementing technology for defense enabling under the DARPA project titled "Applications that Participate in their Own Defense" (APOD). This technology includes most of the defense mechanisms described in section 4.1. Our implementation includes both

- specific examples of defense-enabled applications, and

- a toolkit for configuring an application and its environment to implement a chosen defense strategy.

The defense strategies have been implemented using the QuO adaptive middleware [6]. Our implementation will be discussed in a companion paper [7].

The "Intrusion Tolerance by Unpredictable Adaptation" (ITUA) project [3], also being conducted at BBN Technologies, is exploring two questions about defense enabling:

1. What value does unpredictability add to a defensive strategy?

2. How must a defensive strategy change to handle attackers that can gain application-level privilege?

MAFTIA [8] is an ESPRIT project developing an open architecture for transactional operations on the Internet. MAFTIA models a successful attack on a security domain, leading to corruption of processes in that domain, as a "fault"; the architecture then exploits approaches to fault tolerance that apply whether the faults have an accidental or malicious cause. The MAFTIA architecture appears to be an example of defense enabling.

Other projects have similar goals. The "Survivability Architectures" [5] project aims to separate survivability requirements from an application's functional requirements. "An Aspect-Oriented Security Assurance Solution" is a DARPA-funded project at Cigital Labs that uses aspect-oriented programming to implement security-related code transformations on an application program.

6 Acknowledgements

This work was sponsored by DARPA under contract F30602-99-C-0188.

The authors would like to thank other members of the BBN staff, Ron Watro, Chris Jones, Michael Atighetchi, Tom Mitchell, Ron Scott, Paul Rubel, Craig Rodrigues, and John Zinky, and members of the University of Illinois Proteus team, William Sanders, Michel Cukier, James Lyons, Prashant Pandey, and Hari Ramasamy, for discussions that led to the conclusions in this paper.

7 Conclusion

Defense enabling can increase an application's resistance to malicious attack in an environment that offers only flawed protection. This increased resistance means that an attacker must work harder and take more time to corrupt the application. This, in turn, means greater survivability for the application on its own and an increased chance for system administrators to detect and thwart the attack before it succeeds.

This paper analyzed defense enabling and identified the following as key aspects:

- slowing the acquisition of privilege by the attacker by:
  - distributing parts of the application redundantly across multiple, independently protected, security domains that cannot be attacked concurrently;
  - using cryptographic mechanisms to create a new application-level privilege that is difficult for the attacker to gain even with "root" privilege.
- responding to the attacker's attempts to control resources needed by the application by monitoring QoS and adapting to try to restore it. A variety of adaptation mechanisms were classified.

References

[1] D. Bakken. Middleware. http://www.eecs.wsu.edu/∼bakken/middleware-article-bakken.pdf.

[2] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Comp. Surv.*, 25(2), 1993.

[3] Intrusion tolerance by unpredictable adaptation. http://itua.bbn.com. BBN Technologies and University of Illinois.

[4] S. Kent. On the trail of intrusions into information systems. *IEEE Spectrum*, Dec. 2000.

[5] J. Knight et al. Architectural approach to information survivability. Technical report, University of Virginia, Sept. 1997.

[6] J. Loyall, R. Schantz, J. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp.*, Apr. 1998. Kyoto, Japan.

[7] P. P. Pal, F. Webber, et al. Defense enabling using advanced middleware: An example. In *MILCOM*, Oct. 2001.

[8] D. Powell et al. MAFTIA (malicious- and accidental-fault tolerance for internet applications. In *Int'l Conf. Dependable Syst. and Networks*, July 2001.

[9] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comp. Surv.*, 22(4), 1990.

[10] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[11] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 2nd edition, 1989.

[12] US Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, Dec. 1985. DoD 5200.28-STD.