Survival by Defense-Enabling*

Partha Pal
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
ppal@bbn.com

Franklin Webber
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
fwebber@bbn.com

Richard Schantz
BBN Technologies
10 Moulton Street
Cambridge, MA 02138
rschantz@bbn.com

ABSTRACT

Attack survival, which means the ability to provide some level of service despite an ongoing attack by tolerating its impact, is an important objective of security research. In this paper we present a new approach to survivability and intrusion tolerance. Our approach, which we call "survival by defense" is based on the observation that many applications can be given increased resistance to malicious attack even though the environment in which they run is untrustworthy. This paper describes the concept of "survival by defense" in general and explains the assumptions on which it depends. We will also explain the goals of survival by defense and how they can be achieved.

1. INTRODUCTION

Malicious attacks on computer systems are at the core of security research, and there have been various approaches to deal with the problem. One approach, which sets its goal at *attack prevention*, defines security policies identifying what needs protection and then attempts to implement that protection in hardware and software. This approach has lead to the development of what is known as a trusted computing base (TCB)[17]. Another approach, which is primarily concerned about *attack detection and situational awareness*, has lead to the development of various intrusion detection systems (IDS).

Neither of these approaches is perfect. The TCB is trusted not to violate the security policy itself, and, in most systems, it is also trusted to prevent other, possibly malicious, software from violating the policy. In practice, most computer systems today have no such trusted computing base. 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 already target them, but are unlikely to be redesigned to be more trustworthy. Similarly, although there are many IDSs available today, they mostly work off-line, without any direct runtime interaction or coordination with the applications (and with other IDSs) that they aim to protect. Furthermore, many of the IDSs have questionable accuracy: sometimes they miss real attacks and sometimes they raise false alarms.

In short, attack prevention is not absolute and attack detection is not perfect. We therefore, ask the following question: what, if anything, can be done to tolerate and survive cyber attacks assuming that the environment in which applications will run offers flawed protection and imperfect intrusion detection? In principle, the answer is "close to nothing". 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 in some other way. Although one might try to protect data using encryption and digital signatures that are computationally infeasible to break[16], 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. It may be possible to put various kinds of obstacles and diversions in their path. 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[8]. In any case, the more time an attacker takes, the more vulnerable he is to being detected and stopped by system administrators.

These are the factors that our "defense enabling" approach aims to exploit. We make a distinction between *survival by protection*, which seeks to prevent the attacker from gaining privileges, and *survival by 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 pro-active; de-

*This work is sponsored by DARPA in parts under contracts No. F30602-00-C-0172 and No. F30602-99-C-0188.

NSPW'01, September 10-13th, 2002, Cloudcroft, New Mexico, USA.

fense 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 (i.e., survival), 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 under different circumstances.

2. "SURVIVAL BY DEFENSE" OF CRITICAL APPLICATIONS

"Survival by defense" is not a silver bullet for defending all applications in general. Different applications have different survivability needs and willingness to bear the associated cost. Sometimes, the requirements may conflict with each other. Therefore, a uniform defense for all applications is practically impossible. In our work, we focus on the specific need of a specific type of applications namely, 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[1] 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.

This implies that the defense enabling technology may deploy additional mechanisms in the application's environment, but it is used in the context of defending the critical application. 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 on the application to mask the impact of their exploitation by the attacker. 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[2]. The existence of middleware is an advantage in this context, because it provides a means to incorporate and coordinate the capabilities of the defense mechanisms with the application with minimal modification of the application itself.

An application that does not function correctly is *corrupt*. A corrupt application might deliver bad service or it might fail to deliver any service at all. 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 or in its own implementation cause it to misbehave.

The flaws in the environment and the implementation can be exploited by a malicious attacker to cause a loss of protection that allows the application to be damaged.

In this paper we focus on 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 are random.
- 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.

Given this understanding of corruption, the goal of survival by defense is to *delay* or *prevent* corruption of critical applications. Note that the ultimate way to prevent this kind of corruption is to prevent the attacker from gaining the privilege required to corrupt critical applications, which, as we explained earlier, happens to be the goal of survival by protection. Defense enabling, therefore, can be divided into two complementary goals:

1. The attacker's acquisition of privileges must be slowed down. This issue is discussed in 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.

The first goal makes the protection in the application's environment last longer. The second goal makes the attacker work harder to use newly-gained privileges to corrupt a critical application. Because we have assumed that acquisition of privilege by an attacker cannot be completely prevented or delayed indefinitely, both goals are needed for defense.

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. Section 5 explains that many defense mechanisms will tend to be placed into middleware[2], 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. ACQUISITION OF PRIVILEGE

If privileges could be obtained instantly, the attacker could immediately grab all the privileges needed to stop all application processing and thus deny all service. No defense

[1]Throughout the paper we will use the terms *environment*, *operating environment* or *system infrastructure* to mean mostly the hardware, and networking and operating system software. We will also assume the existence of middleware, a class of software designed to manage the complexity and heterogeneity associated with an inter-networked and distributed environment [2], defined as a software layer above the operating system but below the application programs to provide a common and transparent programming abstraction across a distributed system.

[2]One can, however, add mechanisms like IDSs and firewalls to increase the environment's level of protection, independent of and without the control of critical applications. Defense enabling does not discount such measures, but wants to use their services in application's defense as well.

would be possible against this unlimited attack. Therefore, defense enabling depends on slowing the spread of privilege to attackers to such degree that it renders ineffective the objective of shutting down critical services instantaneously.

In order to prevent quick 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, described in order of increasing privilege (i.e., each of these privileges subsumes all the previous ones), 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).

In addition, we propose to create a new kind of privilege in each domain to impede the attacker's progress towards collecting privileges:

- **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).

Application-level privilege 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. However, 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.

Ideally, one would want the privileges to be discrete and the acquisition process be independant because that will gauarntee an increase in attacker's risk and cost. However, this ideal goal can only be reached partially as explained below. In the sequence of *anonymous*, *domainuser*, *domainadmin* privileges, each subsumes the preveous ones, so if an attacker gains domain admin privilege he does not need to obtain anonymous or domain user privelege. Domain privilege does not imply application level privilege (or vice versa), as explained later, it is possible to obtain application level privilege if the attacker gains domain admin privilege. On the other hand, a malicious intruder will attack a critical application by collecting the minimum privileges needed to damage its integrity or to stop it from providing service. Attackers typically gain new privilege by *converting* from another privelege using some flaws in the environment as opposed to *directly* obtaining the desired level of privilege. In this sense the independance goal is partially achieved.

Using the set of privileges just listed, there are three ways for an attacker to gain new privileges:

1. **Case 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. **Case 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. **Case 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, security domains that are sets of network hosts are a very common situation and the following discussion is applicable in this context. The general idea is to engineer lots of complexities and obstacles in the privilege escalation process so that the attacker work load and likelyhood of triggering detection is increased.

In the first case, the attacker tries 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 careful 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.

In the second case, our objective will be achieved if the attacker is prevented from converting administrator privilege in one domain into administrator privilege in another. This can be done by proper host configuration and administration, and having a heterogeneous environment with various types of hardware and operating systems. For example, hosts in different domains must not respect each other's priv-

ileges. 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.

In the third case, a defense-enabled application must use cryptographic techniques to prevent the attacker from gaining application-level privilege. An attacker having this privilege would be worse than an attacker who becomes a domain administrator because direct attacks on the application cannot be confined to a single security domain anymore: 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 essential for every defense-enabled critical application:

- Application processes must be started with 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 uses sequence numbers to prevent replay.

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[3]. However, someone with domain administrator privilege could gain application-level privilege with enough effort. For example, with administrator privilege, one 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. In practice, however, 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 multiple domains cannot go just as fast as an attack on one single domain. An attack that proceeds sequentially, rather than concurrently, is called a *staged* attack. Defense enabling relies on an attacker using only staged attacks. We can either simply assume that attackers are limited to staged attacks or we can try to make the alternative harder to accomplish. As a practical matter, most attackers will gather privileges sequentially as they explore a system's infrastructure, so this is not an unreasonable assumption. On the other hand, some attacks can be automated and carried out many times in parallel, so in the worst case the attacker can violate an assumption of staging. This worst case can be made less likely by designing an application to use a diverse set of security domains. Diversity means the attacker may need to prepare separate attacks for each kind of domain. It may also be possible to *enforce* staging by configuring firewalls so that an attacker cannot access remote domains at all without first gaining privileges in nearby ones. This paper does not address the issue how to enforce staging but henceforth assumes that only staged attacks are possible.

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.

[3]At a logical level, because the attacker can disrupt the physical communication by cutting the cables, for instance.

4. CONTROL OF RESOURCES

In the traditional approach to computer security, the defender is given additional privilege initially, which is used for setting up static protection both for critical applications and for ensuring that 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 dispute the attacker's control of domains. This is especially important in light of the observation that the attacker and the critical applications compete over system resources: the application needs them and the attacker attempts to take them away from the application. This section discusses the capabilities that can be used to tip the balance away from the attacker. They include:

- **Use of redundancy**: 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 for fault tolerant replica coordination exist[15], [5],[12], [14].

  The fault tolerance problem to solve becomes harder or easier depending on whether the attacker is able to gain application-level privilege. 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 the more expensive protocols that protect against "Byzantine" corruption[3]. If, on the other hand, the attacker does

gain application level privilege, such expensive protocols become necessary. In one of our research efforts, we assume attackers cannot get application level privilege and in another we relax that assumption and explore the use of hybrid-mode fault-tolerance and dynamic switching between tolerating crash and Byzantine failures of application replicas.

Redundancy is not necessarily restricted to redundant processes (replicas as described above) or hosts and security domains. Communication redundancy in the form of redundant bandwidth or alternate network path must also be used by the defense.

- **Monitoring**: As with any other conflict situation, information superiority is an advantage. Therefore, it is important that the defense is aware of incidents in the environment that are related to attacks and their impact on the system resources. Intrusion detection systems (IDSs)[8] can be used, to collect data at the infrastructure level about possible attacks. Data collected at the application 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 a potential 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 is corrupt, possibly because the attacker has gained application-level privilege.

- **Adaptation**: It should be obvious that survival is impossible without adaptation. The consequence of attacker's abuse of the obtained privilege is, almost always, some change in the application's environment ranging from loss or corruption of application components to loss or corruption of resources need by the application. If the application is not able to adapt to the changed situation, the application will not be able to survive. This kind of adaptation may take various forms. If the attacker denies resources to a critical application, for example by flooding communication channels, the defense mechanism may try to adapt to restore the QoS it needs or the application may adapt to live with the reduced resource (thereby degrading the service it offers). 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.

Each of these capabilities are worth separate discussions. We have discussed how to use replication and intrusion detection services to develop adaptive applications that survive certain kinds of attacks in [10]. We will outline our work on use of mixed mode fault tolerance in intrusion tolerance in an upcoming paper [6], and in the next Section, we briefly discuss the various ways defensive adaptation can be used in application's survival.

	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

Note that it is possible for the attacker to defeat or abuse any of these capabilities if he manages to acquire sufficient privilege. For instance, with domain privilege, he can perhaps jeopardize the replication mechanism by shutting down the replication management components that run in that domain. Similarly, with application privilege he can defeat the application level self-checking which affects the application across domain boundaries. We assume that it is not possible to prevent attacks on defense mechanisms that offer these capabilities, just as much as it is not possible to prevent the attacks on the application that these mechanisms aim to protect. However, we have showed that the attacker's acquisition of privilege can be prevented or slowed down earlier, which makes such attacks on the capabilities more difficult and time consuming. Note also that, even though protection of defense mechanisms are not perfect, they raise the bar that an attacker has to overcome in order to successfully stop a critical application from functioning.

5. USE OF DEFENSIVE ADAPTATION IN APPLICATION'S SURVIVAL

One can think of multiple dimensions in which defensive adaptation can be used. The level of system architecture at which these adaptations work is one such dimension. At the top end along this dimension are defensive adaptations involving the application itself: for instance, in the face of an attack the application may find an alternate way to proceed or degrade its service expectations. At the other end along this dimension are defensive adaptations that involve services from the operating system and network level, such as changing the details of how application components communicate among themselves. Between these two are defensive adaptations that manipulate QoS management facilities to obtain the QoS it needs.

In another dimension, adaptations differ according to 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 nor its effects avoided the application can make changes to protect against similar attacks in the future.

Table 1 shows some example adaptations based on the two dimensions described above. The table 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 adaptation involving "honeypots" where an attacker is lured into wasting effort on a decoy.

Attacks can be thought of as two broad kinds:

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

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

This categorization provides the third dimension for classifying defensive adaptations: some work against direct attacks and some against indirect attacks. Direct attacks are countered by the mechanisms working at the application level, plus the use of encryption. An indirect attack might be countered by mechanisms that are at various levels of the system architecture, but generally, lower-level mechanisms are 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, over-consumption of CPU through scheduling and priorities, crashing of a node running an application component by migrating the component elsewhere, and relatively privileged operations can be disabled with access control if there is a high risk that they might be used maliciously.

Whether it can be used for protection from attack as well as for response to attack, or just for response alone, seems to be yet another way to classify defensive adaptation. Mechanisms needed to support some of the defensive adaptation described in table 1, can also be used for protection. For instance, one can start with a high level of self-checking or a very tight access control or a CPU or bandwidth reservation. While this may offer better protection to begin with, some of them come with a high price tag. For instance, an IDS configured to be very sensitive to attack, has significant costs and so needs to be used sparingly. Another case in point is the use of Byzantine tolerance techniques in replication management: although it will offer better protection against corruption it may be impractically expensive to replicate all components of an application in a Byzantine-tolerant mode. This is one of the primary reasons we are investigating adaptive use of mixed-mode replication [6], where only some of the application components will be replicated, and tolerance-modes can be switched between crash and Byzantine. Furthermore, running in "best protection" mode may impede the normal functioning of the system in some cases and so should be used only when necessary. For instance, disabling highly privileged operations may be the safest option, but operators and administrators will need these to perform their tasks. These observations point out the importance of the capability to change between various modes and the associated trade-offs, which are fundamental to our survival by defense approach.

Defensive adaptation, as described so far, is mostly *reactive*, i.e., these adaptations take place in response to some triggers (most often the monitoring mechanisms are responsible for generating the triggers). Defensive adaptation could be *pro-active* as well, in which case the adaptation takes place without any external trigger. For example, a client can periodically change the server it talks to, or a service provider periodically changes the port through which it offers its services. This kind of adaptation is generally appropriate for limiting the attacker's knowledge about the critical system.

A determined attacker can potentially overcome the effects of defensive adaptation if he can easily predict the adaptive response. For instance, consider an attacker whose objective it is to cut off a particular inter-object interaction by flooding an appropriate network segment. If the defensive adaptation responds with establishing a bandwidth reservation, this attack will be thwarted at first. If however, the response is predictable, the attacker may come with a two stage strategy: in the first stage, he will either exploit the reservation mechanism to establish a large reservation for himself or attack the bandwidth management mechanism to make it ineffective. Then in the second stage, he will flood the network. To cope with this kind of planned attacks, some uncertainty needs to be injected in the defense so that the adaptive response is not predictable to the attacker. This uncertainty can come from some secret that is not known to the attacker or can be based upon some nondeterminism incorporated in the adaptation mechanism. In general, uncertainty seems to enhance the value of defensive adaptation, especially in the context of planned and coordinated attacks.

The defense strategy of a particular critical application may involve use of multiple defensive adaptations. Incorporating multiple mechanisms required for individual defensive adaptation 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.

Separating the design of the functional (or business) aspects of the application from the design of defensive adaptation is good software engineering. It is only natural to put the latter into middleware [2], which acts as an intermediary between the application and the infrastructure and provides various services to the application transparently. In addition, advanced middleware such as QuO (short for Quality Objects)[11], provides support for adaptive behavior and QoS awareness which is especially useful for defensive adaptation and monitoring. This way the functionality and the defense mechanisms can be developed in a decoupled manner. Ideally, defensive strategies and mechanisms would be reusable for many different applications. In fact, in most cases, there seems to be a fairly general and reusable mechanism that interpretes or enforces application specific parameters or rules. For instance, the same access control mechanism can be used in different applications with different access control policies. However, there are cases where the mechanisms are more closely tied with the application. For example, self-checking of application invariants will depend on application-specific data structures.

6. ISSUES AND LIMITATIONS

The use of disjoint security domains not only results in redundancy (that there are multiple domains) but also heterogeniety (since gaining access to one domain does not guarantee the same in other domains). This is helpful in terms replicating application components since this makes it difficult to successfully attack multiple replicas. Our use of application level replication is further supported by dynamic adaptation techniques (such as choosing the place to start a new replicca unprredictable).

Our approch to prevent quick spreading of privilege by using security domains and application level privilege can be combined with traditional mechanims like firewalls, even though a firewall can be a single point of failure. However, unlike firewalls, our application level privilege mechanism involves cryptographic keys. We assume that the signatures are unforgeable. We also assume that the keys are distributed a-priori in a trusted manner. The reliance on crypto systems is a limitation [1], but addressing it is beyond the scope of our work. In addition, in the context of a mission critical application the overhead associated with enforcing the application level privilege may impact its performance level.

Defense-enabling makes use of the capabilities of various defense mechanisms. However, it is not simple to combine multiple mechanisms in a defense strategy since different mechanisms may have conflicting goals and assumptions. Although the defense mechanims used in a defense enabled application is coordinated at runtime by the QuO adaptive middleware, the initial work of designing and implementing a defense strategy that involves selection of appropriate mechanism, potential conflict analysis and resolution has to be done manually by an expert.

Finally, defense-enabling relies on the fact that attacks proceed sequentially so that the application have time to make defensive response. As noted in Section 3, there are attacks that may not follow sequential stages. For instance, DDOS attacks involving attackbots or zombies can all attack a target simultaneously. However, if we consider the act of placing zombies as part of the attack we can clearly see a sequence of gaining privilege, inserting processes and then triggering the attack. The problem here is that the most of the preparatory sequences of such an attack can go undetected and can be carried over a large period of time making it hard to correlate. One aproach we are exploring in this regard is to employ autonomous low level mechanisms that perform knee-jerk reaction to anomalies within a host. The idea is that each host will act on its own as soon as it sees an anomaly without the need for a coordinated detection and corelation of events. For example, as soon as it sees an anomalous file in a local disk it may erase that file. This reaction will then be followed up by more coordinated domain or system-wide responses taken at the higher levels. For instance multiple ocurrance of such a mysterious file may lead to isolating that host from its domain.

7. RELATED WORK

MAFTIA[7] 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"[9],[18] 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.

8. CONCLUSION

We are implementing technology for defense enabling under the DARPA project titled "Applications that Participate in their Own Defense" (APOD). The defense strategies have been implemented using the QuO adaptive middleware[11]. The implementation is discussed in detail in [13].

The "Intrusion Tolerance by Unpredictable Adaptation" (ITUA) project[4], also being conducted at BBN Technologies, in cooperation with University of Illinois and The Boeing Company, is exploring two related issues:

1. Tolerating planned and coordinated attacks by making defensive responses unpredictable to the attacker,
2. Tolerating attacks that can gain application-level privilege and take control over application components , by using the services of a hybrid-mode fault-tolerance mechanism.

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.

9. ACKNOWLEDGMENTS

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

10. ADDITIONAL AUTHORS

Additional authors: Joseph Loyall (BBN Technologies, email: jloyall@bbn.com), Ronald Watro(BBN Technologies, email: rwatro@bbn.com), William Sanders (University of Illinois at Urbana-Champaign, email: whs@crhc.uiuc.edu), Michel Cukier (University of Illinois at Urbana-Champaign, email: cukier@crhc.uiuc.edu) and Jeanna Gossett (The Boeing Company, email: Jeanna.Gossett@MW.Boeing.com)

11. REFERENCES

[1] R. Anderson. Why cryptosystems fail. In *Proceedings of the First Conference on Computer and Communication Security*, Nov. 1993. VA, USA.

[2] D. Bakken. *Middleware*. Washington State University. http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf.

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

[4] BBN Technologies. *The Intrusion Tolerance by Unpredictable Adaptation Project.* http://www.dist-systems.bbn.com/projects/ITUA.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX symposium on Operating Systems design and implementation*, Feb. 1999.

[6] M. Cukier and et al. Overview of the ITUA project. In *In Fast Abstracts of the Dependable Systems and Networks (DSN 01) Conference*, June 2001. Goteborg, Sweeden.

[7] IST Programme RTD Research Project IST-1999-11583. *Malicious- and Accidental-Fault Tolerance for Internet Applications*. http://maftia.org.

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

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

[10] J. Loyall, P. P. Pal, R. Schantz, and F. Webber. Building adaptive and agile applications using intrusion detection and response. In *Proceedings of the Networked and Distributed Systems Security (NDSS) Conference*, Feb. 2000. San Diego, California.

[11] J. Loyall, R. Schantz, J. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *1st IEEE Intl Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'98)*, Apr. 1998. Kyoto, Japan.

[12] L. Moser, P. Mellier-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. Eternal: Fault Tolerance and live upgrades for distributed object systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Jan. 2000. Hilton Head, South Carolina.

[13] P. Pal, F. Webber, M. Atighetchi, R. Schantz, and J. Loyall. Defense enabling using advanced middleware: An example. To appear in the Proceedings of MILCOM'01, Oct. 2001.

[14] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM conference on computer and communication security*, Nov. 1994.

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

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

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

[18] C. Wang. A security architecture for survivable systems. Technical report, University of Virginia, Jan. 2001. PhD Thesis.