Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software

Xiaohu Qie, Ruoming Pang, and Larry Peterson
Department of Computer Science, Princeton University
`{qiexh,rpang,llp}@cs.princeton.edu`

Abstract

This paper describes a toolkit to help improve the robustness of code against DoS attacks. We observe that when developing software, programmers primarily focus on functionality. Protecting code from attacks is often considered the responsibility of the OS, firewalls and intrusion detection systems. As a result, many DoS vulnerabilities are not discovered until the system is attacked and the damage is done. Instead of reacting to attacks after the fact, this paper argues that a better solution is to make software *defensive* by systematically injecting protection mechanisms into the code itself. Our toolkit provides an API that programmers use to annotate their code. At runtime, these annotations serve as both sensors and actuators: watching for resource abuse and taking the appropriate action should abuse be detected. This paper presents the design and implementation of the toolkit, as well as evaluation of its effectiveness with three widely-deployed network services.

1 Introduction

Denial-of-Service (DoS) attacks are a major source of concern in the Internet. Unlike security break-ins that obtain privileged access, DoS attacks are designed to consume a disproportionate amount of resources on the target system by exploiting weakness in the network software. When successful, such attacks make the system unavailable to well-behaved users.

Common defenses against DoS attacks include using firewalls and Intrusion Detection Systems (IDS) to monitor network links for offending traffic, as well as applying software patches to fix known vulnerabilities. However, such defensive practices burden the system administrator with making sure all systems have the up-to-date patches installed and all firewalls are properly configured. To make matters worse, even after a new attack is recognized, it is not until the vulnerabilities exploited by the attack are determined that a patch can be developed.

We observe that many DoS vulnerabilities can be attributed to the separation of software functionality and protection. When developing software, programmers primarily focus on functionality. Protection from attacks is often considered the responsibility of the OS, firewalls, and IDS, and thus not an immediate concern. As a result, many vulnerabilities in the code are not discovered until the system is hit by an attack that exploits the weakness, that is, after the damage is done.

Instead of reacting to attacks, we propose a new approach to DoS protection: *defensive programming*, by which we mean programmers embed *general* mechanisms into their software to provide *systematic* and *proactive* protection against DoS attacks. Ideally, defensive software guarantees availability even under a previously unknown DoS attack. An important aspect of this approach is that it should be designed to thwart common DoS attack characteristics; programmers should *not* have to scan their code for a specific implementation vulnerabilities and fix them, as they do when writing a software patch.

Towards this end, this paper describes our experience developing mechanisms to help programmers systematically build robust software. The key idea is to insert annotations that monitor and control the execution of the program at runtime. These annotations serve both as sensors that detect anomalies and actuators that change the control flow of a program when they detect that defensive measures are necessary. The advantage of annotations is that they allow us to adjust the program's behavior at a very fine granularity, thereby making it possible to confine the damage of an attack without negatively affecting other aspects of the program.

We have developed a toolkit consisting of a set of annotation primitives, a runtime library, and a set of compiler extensions. As a means of specifying a resource management policy, a programmer inserts annotation primitives into code so that the annotations mark where resources are acquired/released/consumed, where the program branches into independent functionalities, and what principals are holding resources. The compiler extensions check consistency among annotations by analyzing the control flow graph of the program and generating necessary code to be executed at annotated points. At runtime, appropriate monitor and control functions are invoked as control flow passes through these annotations.

The toolkit helps programmers reason about DoS problems in a more structured way. Rather than focus on implementation details, they are asked to identify the services provided and the resources consumed by their program at a high level. For example, if the programmer annotates a certain function as performing an identifiable service, the toolkit will confine a DoS attack on that service to requests of the same service, rather than letting the attack bring down the program as a whole. The flip-side, of course, is that the toolkit is not a panacea. Like any mechanism, the effectiveness of the toolkit depends on whether a good defensive policy can be specified, which is ultimately the responsibility of the programmer.

The paper makes two contributions. First, it studies the general question of how to develop defensive code that protects itself from DoS attacks. In the process, the paper identifies a class of attacks that exploits a vulnerability existing in many network servers, but that has not received attention in the literature. Second, it describes a specific mechanism—the annotation toolkit—that evolved from this study. We have implemented the toolkit in Linux, and demonstrated how to annotate widely deployed software, including the Linux IP protocol stack, the Flash web server [10], and the Linux NIS servers [8]. Our experience shows that we can significantly improve the robustness of software against DoS attacks with relatively low programming effort.

2 Related Work

Our approach to writing defensive code draws on previous research in several areas. This section explains how our work fits in this larger design space.

2.1 Intrusion Detection Systems

Anomaly detection uses statistics of normal behavior as a baseline, and treats changes in these patterns as an indication of an attack. Researchers have demonstrated that examining the sequence of system-calls made by an application is a viable approach to detecting security violations due to bugs in the program (mainly buffer overflows) [6, 14, 18].

However, current anomaly detection techniques have difficulty detecting resource-exhausting attacks, because a DoS attacker can request the same service as a legitimate user. Our approach has the flavor of anomaly detection, but with a focus on resource usage rather than security. Since the target of a DoS attack is some resource on the victim system, we instrument the program to look for irregularities in resource usage and actively participate in resource management. In a way, we do not have to distinguish DoS attacks from other activities, the rationale being that as long as resources are properly managed, the damage any DoS attack can cause is limited.

2.2 Performance Monitoring

Resource-exhausting DoS attacks often cause performance degradation on the target, making it possible to detect such attacks by monitoring the profiling data. In general, however, we found profiling-based detection insufficient for the following reasons. First, profiling does not cover all the important aspects of a program's behavior. The target resource of a DoS attack is not necessarily CPU cycles; sometimes it can be application-level objects. Second, getting the average behavior from profiling data is not enough because even perfectly legitimate users can deviate significantly from the average without attacking the system. To infer the behavior distribution from profiling data is a hard problem that does not have a good solution for the general case.

In order to collect comprehensive data for analysis and extract meaningful information from the data, it is necessary to know what resources a program consumes, as well as where and how they are being used. Our annotation interface allows a program to provide such information.

In performance assertion checking [11], the original program is instrumented to generate an execution log, which is then checked offline for performance violations. The assertions and logging facilities are the counter parts of our resource sensors. Being independent of the original program, assertions can be declared in a more expressive language. In contrast, our resource sensors and actuators are part of the original program being annotated, monitoring resource usage and changing the program control flow at runtime.

In addition, our goal is not only detection, but also protection. Since an appropriate defensive action is highly dependent on the functionality and architecture of the program, the action has to be specified at the source code level. Watching profiling data can sometimes tell us the system is being attacked, but without a defense mechanism built into the program, the only available response is to kill the victim process, which is a DoS attack in its own right.

2.3 Static Code Analysis

There has recently been much work in automatic detection of software errors and security bugs through static code analysis. Recent work by Engler *et al.* [4, 5] introduced the technique of *meta-level compilation*. The idea is that the software must obey certain rules for correctness, such as "kernel code cannot call blocking functions with interrupts disabled" and "message handlers must free their buffer before completing". System programmers specify the rules in a high-level language, and an extensible compiler then applies the rules throughout the program source to check for violations. Meta-level compilation is very successful in finding errors in OS code, as well as a wide range of security bugs using rules such as "do not dereference user pointers without checking validity". The authors found several DoS possibilities in the kernel code they examined, but the result is limited to a special case in which an attacker controls the iterations of a kernel loop.

Static analysis alone is not sufficient for detecting DoS attacks since such attacks do not necessarily rely on software bugs. It is often the cumulative pressure on resources that puts a system in peril, even though the software itself is bug-free. Thus, besides examining how the software is implemented, we must also watch how it is *executed*. Such information can be only collected at runtime with additional application or OS support. Previous work in detecting race conditions in concurrent programs [12] seems to support this point of view. Our approach differs from previous work of static analysis mainly in that we check for possible "rule" violations at runtime, with a focus on resource usage.

2.4 OS Mechanisms

There has been an ongoing effort to build new OS mechanisms and specialized OSs to provide service differentiation and guarantees. For example, Resource Containers [3] are an abstraction that takes over the process' role as the primary resource principal. It allows multiple cooperating processes to bind to the same container, as well as a process to change its resource and schedule binding dynamically when it executes on behalf of another activity. The Scout operating system [9, 15] uses a similar abstraction—the path—as the primary resource and schedule principal. Both systems have been shown to be able to defend against certain flooding DoS attacks. The improvement results from more accurate resource accounting and service isolation.

An important contribution of resource containers is the separation of resource principals and execution domains, but as an OS approach, resource management policies are ultimately enforced via process scheduling among execution domains. In case an execution domain multiplexes among a set of resource principals, resource containers reduce to a passive accounting facility. However, many functionality-rich services, such as web servers and routing daemons, are single-process-event-driven. *Intra-process* protection is more important for these applications, since we do not want to penalize the entire process when just one of its functions is being abused. This calls for a finer-grain resource protection than what can be provided by an OS approach. Using annotations inserted by the programmer to monitor and control the execution path within a process, our approach offers a finer-grain protection than OS approaches.

Another system-level approach, SEDA [19], proposes a programming model in which a program is divided into stages and each stage enforces its own resource management policy by controlling threads running in that stage. This model differs from the traditional process-based resource protection in that resource allocation not only depends on the process, but also on the stage in which the process is running. From this perspective, our approach is similar to SEDA. On the other hand, SEDA is not intended for DoS protection, and does not protect resources that cannot be protected by scheduling.

Finally, our toolkit is intended to improve the robustness of existing software. Annotating code is more programmer-friendly than imposing a new OS architecture or abstraction, which often requires rearchitecting code. This is especially true with Scout and SEDA.

3 DoS Attack Characterization

Researchers have studied many DoS attacks [13, 7]. What is lacking, however, is an analysis of their common characteristics: what they attack and how they attack it. Such a characterization would help us understand the signature of DoS attacks, and shed light on how to systematically and proactively write defensive software.

There are several well-known attacks on network software, including the ICMP flood attack (send a large number of ICMP echo packets at the target), TCP SYN attack (flood the target with connection-open requests), and Christmas Tree packets (overwhelm a target with packets that have exceptional bits turned on in the header—e.g., IP options—dictating the packet receive special processing). A less well-known attack, which we refer to as route cache poisoning, involves an attacker flooding a router with packets carrying a sequence of nonsensical IP addresses (e.g., "1", "2", "3", and so on), thereby blowing the router's first level route cache. This causes the router's control processor to spend all its time building new microcode and loading it into the switch engine. This happens at the expense of the router responding to its neighbors' routing probes, which causes the neighbors to believe the router is down.

These examples illustrate that DoS attacks abuse a legitimate service by sending it a large volume of requests, suggesting that rate limiting [17] and load conditioning [19] would be an effective defense. However, DoS attacks can also be carried out in a way that renders rate limiting strategies ineffective. The following example illustrates this possibility.

3.1 Slow TCP Attacks

Many TCP-based services follow the request-reply paradigm. Since a server must set aside resources while a client request is being processed, it is possible to exhaust the server's resource by manipulating the operation of TCP. The idea behind the attack is for the client to make the TCP connection as slow as possible. This simple idea can be realized in three different ways.

First, a client can send the request very slowly. Since

TCP is a byte-stream protocol without record boundaries, the server cannot interpret the client's request until all the data is received. Suppose a request contains 2000 bytes, and the TCP MSS is 1000 bytes. Under normal operation, the client would send the request in two packets. If, instead, the client sends the request one byte at a time, which does not violate any protocol and application requirements, it would take 2000 RTTs before the server can start to process the request. The client can insert additional delays between packets to further extend the duration.

Second, once the server starts to send results back, the client can read the data very slowly. The server side TCP would interpret the closed TCP advertised window in the acknowledgment packet as a signal that the client application is temporarily busy, thus pause sending.[1] The server will not be able to send more data until the window is opened again. Thus by abusing TCP's flow control mechanism the client can pace the rate of data sent by the server.

Third, the client can acknowledge the response very slowly by pretending the packet was lost. Without seeing an acknowledgment, the server will retransmit. Similar to the slow receiver, the client can pace the sending rate of the server by controlling when to acknowledge a packet. In this scenario, the client abuses TCP's reliable transmission feature.

One target of the Slow TCP attack is web servers. Being a slow sender, an attacker can construct an extremely long HTTP request (e.g., copy the header "`User-Agent: Slow TCP Sender \r\n`" 5000 times) and send it at a very low rate (e.g. 1 byte every 50 seconds). Being a slow receiver or ACKer, an attacker just requests a big file then nibbles at the server's output. The goal of the attacker is to keep the connection alive as long as possible. Since the number of concurrent connections a web server can maintain is limited, given sufficient number of slow attackers, the server's available connections will be exhausted, and all subsequent requests will be denied.

We verified this idea experimentally by implementing a HTTP request generator that uses slow TCP, and tested it against two popular web servers: Apache [2] and Flash [10]. The attack proves to be extremely effective. Despite the fact that TCP has a keep-alive timer, the Linux TCP implementation limits the number of retransmission attempts to 12, and both Apache and Flash have built-in mechanisms to time-out idle connections, all three forms of slow attacks are able tie up a connection for several days, causing the servers to disappear from the net. We were also able to attack NIS servers in a similar way.

In general, we believe such attacks are not limited to TCP servers. For example, an attacker could disable a firewall that provides NAT or Proxy services by repetitively sending packets from all available ports to a random set of destinations. Once the translation table on the firewall is filled up, other users are effectively cut off from the rest of the Internet.

[1] After some time, the server TCP will send a 1-byte packet to test if the client has consumed any data.

3.2 Attacks Revisited

When characterizing DoS attacks, it is helpful to distinguish between two types of resources: *renewable resources*, such as CPU cycles, the bandwidth of network, disks, and buses; and *non-renewable resources*, such as processes, ports, buffers, PCBs, and locks. To attack a renewable resource, the attacker continually consumes the resource so that legitimate services do not receive enough of the resource over time. This is usually achieved by flooding the server with massive number of requests in order to keep the target system busy. In contrast, if the target resource is non-renewable, the attacker tries to acquire as many resource as possible and does not release them. This form of attack does not require flooding to make the target busy.

In the rest of the paper we denote an attack targeting a renewable resource a *busy* attack, and an attack targeting a non-renewable resource a *claim-and-hold* attack. However, we note that some attacks cannot be clearly placed in one category. For instance, the target resource of SYN flooding attack is half-open connections, which is a non-renewable resource, but to exhaust this particular resource, the attacker must keep the system busy with a flood of new requests. In another example, router cache poisoning succeeds when the router's CPU is overwhelmed, thus it is a busy attack, yet it works by directly attacking the route cache, which is a non-renewable resource.

These "exceptions" are not special cases, but in fact, phenomenon due to the duality between busy and claim-and-hold attacks. Often in mending one vulnerability, we open the system to another vulnerability. For example, the Apache web server sets a limit of 150 connections to protect itself from runaway resource consumption, yet by enforcing this limit, connections become a "scarce" resource and the program is potentially vulnerable to claim-and-hold attacks. On the other hand, to protect non-renewable resources, the system must perform a recycling function when the resource becomes unavailable. This function itself could become an accessory in a busy attack if it is not resource-controlled. This is the weakness exploited by the route cache poisoning attack. Clearly, a general defense mechanism must protect the system from both types of vulnerabilities at the same time; watching only one type of attacks is not sufficient.

4 Defensive Strategies

Our overall strategy is to separate resources among activities in a program along two dimensions. For renewable resources, we balance resource usage among program functionalities, thereby confining the impact of an attack to the individual service being attacked. For non-renewable resources, we identify principals that hold non-renewable resources and reclaim resources from principals that are not making minimal progress. These two aspects of our strategy are discussed in turn.

4.1 Busy Attack Defense

The strategy is to balance resource usage among program functionalities, thereby confining the impact of an attack to the individual service being attacked. Towards this end, we introduce the concept of *service* and propose a resource control mechanism with actuators at service entries and sensors at resource access points.

4.1.1 Services and Resources

We define a *service* to be a program component that provides an *independent* functionality. Each service, in turn, consumes some amount of renewable resources. Figure 1 shows the conceptual model of a server program divided into services. Client requests are served by different services, as they execute a code path through the program, and multiple services share various resources.

There is often a clear correspondence between services and program code paths, and in many cases, a service is implemented by a particular function and associated subroutines. For example, in the Linux kernel, each ICMP service is handled by a distinct function with name `icmp_<service>` (e.g. `icmp_echo`). Thus, a program can be divided into services according to code paths. To expose the service structure of a program, we ask programmers to annotate the service entry functions in their programs. We have also built a set of compiler tools to help user check coverage and consistency of service annotations.

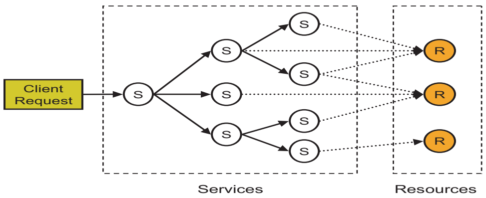


Figure 1: Service View of a Program

We assume each service is performed by a function. When this is not the case, the programmer must extract the part of code that performs the service and wrap it in a separate function. Our experience with the Flash web server and the Linux TCP/IP code suggests that there are few places we need to do the extraction and all of them are straightforward. The benefit of marking functions instead of arbitrary code regions as services is that the user need only annotate service entry points. Our compiler can then automatically annotate the corresponding service exit points, thereby reduce the overall programmer workload and the chance of inconsistent marking. Also, the service hierarchy structure is clearly represented by the function call graph.

Services can be disjoint or nested. For example, in the Linux IP stack (Figure 2), `TCP-recv` and `UDP-recv` are disjoint services, while the service of `IP options` processing is nested inside IP processing. Nested services allow the programmer to divide a coarse-grain service into finer-grain sub-services. Dividing services in this way has the advantage of confining the damage of an attack within a smaller range. When a nested service tries to over-use some resource, action is taken only on the inner-most service that directly uses the resource, for fear that doing anything to the parent services may over-penalize sibling services. For example, if we further divide the service of IP option handling into a sub-service for every type of IP option, then when the code dealing with one type of option is attacked, all other IP options can be still be handled normally.

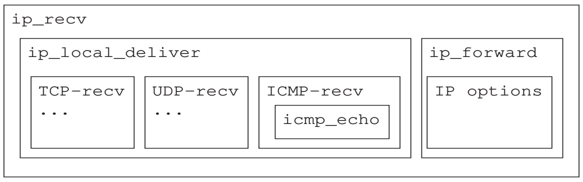


Figure 2: Services in Linux IP Stack

As services correspond to code paths, we can control resource usage of a service by rate-limiting execution on its code paths, especially the "expensive" ones. For example, the Linux kernel checks a rate limit when deciding whether to send out an ICMP packet. We can view the act of changing from one execution path to another, based on resource usage, as intra-process "scheduling" among services. However, since we do not know which code path will be attacked, and it is hard to precisely tell how expensive code paths are, there are two interesting questions in rate-limiting code paths: 1) where to place sensors that monitor resource usage and actuators that change the program execution path; and 2) at what rates code paths should be limited, or how to decide whether or not to switch out of the current code path each time execution reaches the actuators.

4.1.2 Sensors and Actuators

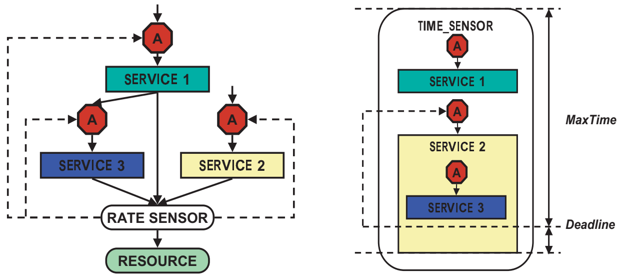
We need a systematic way to place sensors and actuators in the program, because placing them in an ad hoc way may leave holes to be exploited—the code path being attacked might not have an annotation on it. On the other hand, we want to minimize the number of annotations, especially actuators, because switching out of a code path needs to be handled in a program-specific way, and it takes programmer's effort to write such a handler.

Rate limiters found in existing software, such as the Linux kernel, are actually a composite component that consists of both a sensor and an actuator: the sensor monitors the execution rate on the code path, and the actuator deflects the execution to another code path when the rate limit is violated. This approach works well because we know which potential attacks we want to defend against and therefore can put rate limiters on the right code path. In our case, we do not assume that we know about any particular attack. With this different assumption, we found that actuators and sensors need to be placed at different locations in the program, in order that (1) actuation happens at the right place, and (2) resource usages to be properly limited. The following discusses the placement of actuators and sensors, in turn.

For actuators that control the execution path, we argue that service entry points are the right place for them to be placed. This is for three reasons. First, a service is the unit of fault isolation, and activities within the same service share fate. Therefore, it is better to not begin processing a service request if it cannot acquire enough resources to complete. Second, it is easier to abort or delay processing a service request at the entry point than in the midst of processing. Third, each service needs only one actuator, thus the total number of actuators depends only on the number of services.

A potential trade-off here is that sometimes at the service entrance we may not be able to precisely predict whether a request can get enough resource. However, in all busy attacks we know of, a service must be invoked at a high rate in order to exhaust system resource. Therefore, the effect of this inaccuracy is minor, because it matters only when the service is about to reach its resource quota. In other words, rate precision is not so important in DoS defense, as we are not making QoS guarantees. Finally, the programmer may define finer-grained services to achieve better precision.

For sensors that monitors resource usage, we argue that they should be put at resource access points, for example, where a system call is invoked to transmit a packet. If we were to put sensors together with actuators at service entries, it would require much effort and experience to set an appropriate rate limit for each service because it is unclear how service rate limits would map onto actual resource usage. As we try to set the limits for services before knowing which service will be attacked, there is a risk of being either too conservative or too optimistic. Also, the choice is often host-specific



(a) Rate Control (b) Time Control

Figure 3: Managing Renewable Resources

and cannot be easily shared or reused. In contrast, it is straightforward to measure the resource usage at the point the resource is accessed, and it is relatively easy for the programmer to specify an overall rate limit for each type of resource.

Taken together, the sensors monitor both the overall resource usage and usages by individual services, thereby affecting admission decisions at actuators placed at service entry points. (See Figure 3(a)). Actuators control admission to any service that tries to consume disproportional amount of resource. Further details about the actual mechanism is discussed in Section 5.3.

4.1.3 Controlling Continuous Resource

The discussion to this point assumes that resources are always consumed at particular locations of the program. We further distinguish between two types of renewable resources: *discrete* resources, which include almost all renewable resource except CPU time (e.g. network/disk bandwidth); and *continuous* resources, which include CPU time. Unlike discrete resources, CPU time is spent continuously as the program executes, so we can no longer monitor resource on some particular code paths. Therefore it needs to be managed differently.

There are mainly two questions: how to detect CPU overload and how to locate the service being exploited. Our approach is to ask the user to specify time limits on some high-level functions for each invocation, and we control admission to the downstream service that violates the deadline, as shown in Figure 3(b). Again, more details are given in Section 5.3.

4.2 Claim-and-Hold Attack Defense

In order to consume renewable resources, the attacking activity must be *active*, i.e., executing code on the CPU. This observation has greatly simplified our solution to defend busy attacks—basically we need to control the execution frequency and duration of different code paths. Protecting non-renewable resources,

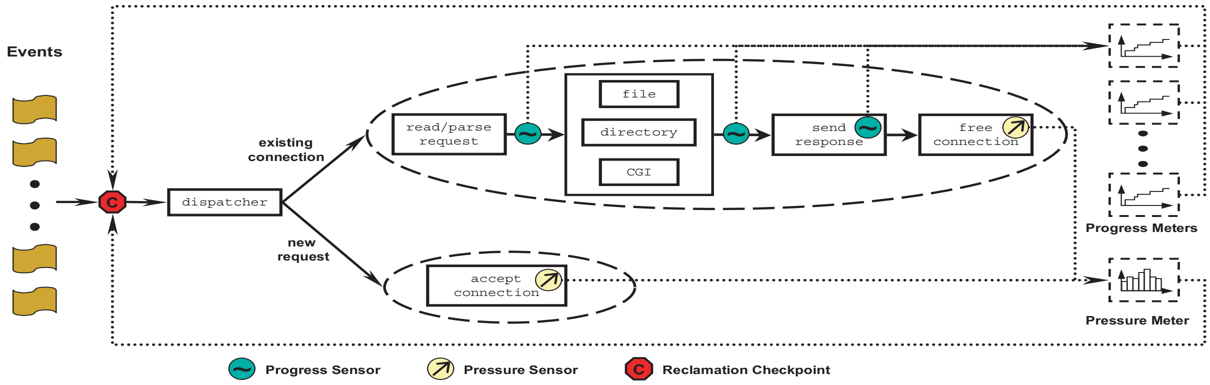


Figure 4: Managing Non-renewable Resources in an Event-driven Web Server

however, is a different story. Attackers holding the resource do not necessarily have to remain active once the resource is acquired.

Protecting non-renewable resources is essentially a process of specifying a replacement policy: when the resource becomes exhausted, which ones should be reclaimed. Resources can be reclaimed either periodically or when some event indicates recycling is necessary. Thus, the problem boils down to one of deciding: (1) what resources to reclaim, and (2) when to reclaim them. We introduce two metrics—*progress* and *pressure*—that characterize these two aspects of a replacement policy, respectively. Our defense strategy involves annotating a program with sensors and actuators that set and react to these two metrics.

4.2.1 Progress and Pressure

In the Slow TCP attack against web servers, the resource in question is the server connection. Neither Apache or Flash implements an explicit replacement policy. When connections are exhausted, the server simply rejects new requests. The connection resource is returned when the client request is completed. It is also reclaimed by timers. In Flash, there are two build time parameters, `CGI_TIMELIMIT` and `IDLEC_TIMELIMIT`. The former caps the maximum running time of a CGI program forked by a client request, and the latter controls the maximum period a client can be idle. When either limit is exceeded, the connection is dropped and resources associated with this connection are freed.

The weakness of this simple mechanism lies in the fact that an attacker can trick the server into thinking it is still in the middle of a request, thereby holding resource without triggering the timers. Alternatively, to guarantee availability, we could choose to tear down the oldest connection when the connection table becomes full. The problem with this approach is that it is biased against clients on a slow link or those downloading a large file.

A better solution is to measure how well a client is making use of the resources it has acquired, and combine this information with other metrics such as age. A client should be allowed to hold resources longer than others, as long as it has a good reason. We use *progress* to denote such a metric. The exact form of progress depends on the resource and application in question, but in general, a proper progress metric should reflect how the principal holding a resource is making use of it. Progress is expected to increase proportionally with time. In the web server example, how many bytes the server has sent to the client could be used to construct the progress metric.

A replacement policy also has to specify when to reclaim resources. Since recycling itself could be an expensive operation, uncontrolled invocations also open up the possibility of busy attacks, which is what we saw in the route cache poisoning attack. We define a *pressure* metric to control the invocation of the reclaim function. Intuitively, resources should be recycled when the pressure on it exceeds a certain threshold, which could be caused either by too many clients requesting the resource, or no clients releasing the resource.

Programmers can develop other metrics tailored to the application. As a general toolkit, we currently only support interfaces to keep track of progress and pressure, on top of which a variety of policies can be built.

4.2.2 Placing Sensors and Actuators

Figure 4 illustrates how sensors and actuators can be placed in an event-driven web server such as Flash.

Sensors are inserted into a program to track (record) progress in two ways. If the principal in question generates output of some kind, the unit of the output is a natural measure of progress; e.g., one can annotate a program with a progress sensor that records how many bytes have been read or written, how many packets have been forwarded, and so on. In a second scenario, an entire task can be broken into stages, where progress is recorded when the task moves from one stage to the next. For example, the Flash web server breaks client request processing into three stages: request reading and parsing, back-end processing, and result sending. Some stages can be further divided depending on the operations required by a particular request (e.g, requesting a static page vs. dynamic content). A stage is represented by a unique "handler" associated with a connection. In this example, progress sensors can be placed where the connection handler is changed.

It is usually obvious how to insert sensors into a program to track pressure: there are often well-defined points in the program where non-renewable resources are accessed; e.g., inside resource allocators and deallocators. Pressure sensors can be placed at these points. Some abstract non-renewable resources are not accessed via an explicit function interface, in which case we need the programmer to annotate the points at which the resource is acquired and released.

Turning to the actuator side, there is a single reclamation actuator that is a function of both metrics: it decides to reclaim resources if the pressure metric is greater than some threshold, and should this be the case, it uses the progress metric to decide which instance of the resource to reclaim. Reclamation actuators are placed in two types of locations. First, the trigger role of the pressure sensor suggests that a reclamation actuator should be placed immediately after a pressure sensor. In fact, we we envision a combined pressure annotation marking the point where resources are claimed and released.

In addition, however, pressure also needs to be examined periodically, as it could build up even in the absence of activity. This implies that we also need to insert a reclamation actuator—which we call a *reclamation checkpoint* to distinguish it from the combined pressure sensor/actuator—that is periodically visited by the control flow. For most server programs this is not a problem as they are iterative by nature. For example in Flash, we could place such an actuator inside its main event loop, as shown in Figure 4. An important issue however, is that when an action is taken, it must not leave the server in an inconsistent state; e.g., not free all resources associated with an activity, or continue to reference a principal that is no longer valid due to the reclamation. We do not have a general solution to the problem, except that by imposing transaction semantics the risk of inconsistency can be reduced. In other words, the checkpoint should be placed outside all functions that are considered atomic.

Finally, when placing a reclamation checkpoint we need to consider how often it is visited by the program control flow. If the interval is not properly bounded, we effectively lose control on the resource. One way to preserve granularity is to use the techniques presented in the previous section, such as the time-sensor, to limit the branches leaving the checkpoint. But under extreme situations, for instance an attacker causing the program to enter an infinite loop, we could still lose control. We considered other alternatives, such as using a timer signal to perform resource checking, but it is extremely hard to perform resource reclamation in a signal handler while still guaranteeing such operations do not lead to inconsistencies. We consider this as one limitation of intra-process protection—sometimes we need to depend on inter-process protection provided by the OS. In other words, there is a trade-off between absolute control and preserving the original program structure.

5 Annotation Toolkit

This section describes our annotation toolkit in detail, focusing first on the annotations themselves, and then on the underlying implementation.

5.1 Renewable Resource Management

The toolkit includes annotations that are used to denote admission control upon service entry, plus annotations that serve as sensors for monitoring rate and time limits. We consider each in turn.

• SERVICE_ADMISSION(min_rate)

The user marks a function as a service entry point, specifying the minimum rate at which that service is allowed to proceed. For example, the following is from the service that satisfies cold cache requests in the Flash web server:

```
SRCode
ProcessColdRequest(httpd_conn* hc)
{
  if (!SERVICE_ADMISSION(3))
    return SR_PLEASE_TRY_AGAIN_LATER;
  /* rest of the function ... */
}
```

This annotation does not directly change the execution path of the program, but returns a hint on whether the service should be admitted based on its resource usage, allowing the program to (1) do necessary cleanup before aborting, (2) delay servicing the request, or (3) ignore the hint. The annotation takes parameter *min_rate* and always returns 1 when the service is invoked below the minimal rate, regardless whether the service has used up its resource quota. This allows users to guarantee service rate for some important services under resource contention.

• RATE_SENSOR (max_rate, weight)

This annotation is used to specify the maximal weighted rate for a particular code path. For example, in order to rate-limit the packet and byte rates of ICMP, we may annotate the code with the following lines before ICMP pushes a packet to IP:

```
if (!RATE_SENSOR (sysctl_icmp_max_msg_rate, 1))
  icmp_msg_rate_violation++;
if (!RATE_SENSOR (sysctl_icmp_max_byte_rate, msg_size))
  icmp_byte_rate_violation++;
ip_build_xmit(...);
```

RATE_SENSOR can be placed any where in the program, unlike SERVICE_ADMISSION which must be put at function entries. It returns a hint on whether the current measured rate of the code path is within the specified maximal rate. However, it is completely legitimate for programmer to ignore the hint (as in the example above) if the limit is not strict. This is because the annotation sends feed-back to the service admission point, thereby eventually limiting resource usage to the specified rate.

- TIME_SENSOR (max_time)

This annotation is used to monitor the execution time of a function (and its subroutines) on each invocation. It is applied on functions in the same way as SERVICE_ADMISSION. For example, to control the execution time of an event handler in Flash web server, we extract the invocation of the event handler into a separate function and annotate the function with TIME_SENSOR so that admission to services invoked by event handlers will be bounded by the time limit.

```
static void LaunchHandler(...)
{
  TIME_SENSOR(handlerTimeLimit);
  handler(tempConn, i, do_what);
}
```

5.2 Non-renewable Resource Management

The toolkit also includes a set of annotations that both demark the allocation and freeing of non-renewable resources, and check to see if resources need to be reclaimed.

- RESOURCE_DECL(resid)

This annotation declares a non-renewable resource that needs protection, where *resid* is a unique identifier. The annotation initializes a data structure to represent the resource. This annotation should be placed in the initialization part of a program.

- RESOURCE_ACQUIRED(resid, p, amt)
- RESOURCE_RELEASED(resid, p, amt)

These two annotations take an opaque pointer and the amount of resource being accessed. The pointer serves to identify the principal; it is usually an application-specific data structure. The annotation also records the timestamp of the operation in order to calculate the duration of resource being held by the principal.

- PRESSURE_SENSOR(resid, s)

This annotation records pressure on the resource caused by discrete events, such as a new request being denied due to the lack of resources. The second argument can be used to express the severity of the situation.

- RESOURCE_UNAVAILABLE(resid)
- RESOURCE_AVAILABLE(resid)

Some applications disable new requests as soon as the resource is used. In this scenario, pressure cannot be tracked in a discrete fashion. Instead, pressure accumulates continually over time when no resources are released. These two annotations are used in such situations.

- PROGRESS_SENSOR(resid, p, prog)

This annotation updates the progress metric of a principal by *prog*. The use of the opaque pointer *p* should be consistent with that in RESOURCE_ACQUIRED and RESOURCE_RELEASED.

- RECLAMATION_CHECKPOINT(resid, cb, min_pres, min_prog)

This annotation is the actuator that performs resource recycling. By default, it takes resources back from the principal making the least progress. Programmers can configure the operation with two additional parameters: *min_pres* specifies that actions should be taken only when the pressure exceeds certain threshold; *min_prog* restricts the actions to be taken only upon principals making less progress than the parameter. By setting different thresholds, a programmer can control the frequency of recycling and give principals that have already made significant progress an allowance to finish the task. Programmers also need to specify a callback function *cb* that is invoked by the actuator. It should free resources associated with a principal (identified by the opaque pointer), but can also be used to log activity for offline analysis.

5.3 Implementation Details

Each annotation is implemented as a C-macro, and is linked with an instance of a corresponding data structure. Key data structures in our toolkit include service, rate sensor, time sensor, resource, and principal, with each maintaining a different set of counters.

A service structure contains a rate counter for service entry rate so that it can tell whether the entry rate is below the minimal rate given in the annotation. It also contains flags to indicate resource or time limit violation by the service. The rate counter is reset to zero at the end of every period (a period lasts for one second in our prototype). The violation flag is also adjusted periodically.

To account resource usage of services, global variable *current_service* points to the service currently being executed. As services can be nested, the variable is updated on each service entry and exit. (Our compiler extension inserts service exit calls corresponding to SERVICE_ADMISSION annotations.) The following gives pseudo-code for service admission and exit:

```
do_service_admission (svc_id, min_rate) {
  if (at the end of period)
    adjust rate and time violation;
  update service entry counter;
  check_deadline();
```

```
  set current_service to svc_id;
  if (service within min_rate || there is no violation)
    return 1;
  return 0;
}

do_service_exit () {
  check_deadline();
  set current_service to parent service;
}
```

The rate sensor structure contains a rate counter *for each service* that uses the rate sensor and a counter for the overall rate. In addition, it maintains a shared rate limit for services: whenever a rate counter of any service exceeds the shared rate limit, the service is marked with a rate-limit violation flag, and its subsequent admissions will be rejected until the end of the period (with the exception of services that are admitted because they are below the minimal service rate). The shared rate limit is adjusted at the end of each period with *additive increase / multiplicative decrease (AIMD)* depending on whether the overall rate exceeds the given limit on the sensor. Below is the pseudo-code for rate sensor:

```
do_rate_sensor(rate_id, max_rate) {
  if (at the end of period)
    adjust shared limit AIMD (total rate counter, max_rate);
  update per service and total rate counters;
  if (per service counter > shared limit) {
    set rate violation on current_service;
    return 0;
  }
  return (rate_counter(rate_id) <= max_rate);
}
```

Adjusting the shared rate limit dynamically allows more flexible rate control than computing the limit with min-max algorithm, which assumes that every service obeys the shared limit. The programmer may allow some service to use more resources than the common share—by overriding it with minimal service rate or ignoring the result of `SERVICE_ADMISSION` —but the shared rate limit is adjusted to a level so that the overall rate still matches the specified limit. This allows users to make application-specific decision on resource allocation other than purely "fair" sharing.

Like the `SERVICE_ADMISSION` annotation, the scope of a `TIME_SENSOR` annotation includes the current function and all its subroutines. At entry `TIME_SENSOR` computes and stores a deadline in global variable *current_deadline*. When `TIME_SENSOR` is applied in a user-space process, the time-stamp is obtained by getting process usage time (which is process time plus system time on behalf on the process) in order to exclude the impact of process scheduling. (In contrast, `SERVICE_ADMISSION` and `RATE_SENSOR` uses wall time.) Within the scope of time-limit, the current time is compared against *current_deadline* (see the pseudo-code for `check_time_limit` below) at each service entry and exit. If the deadline is missed, the current service is marked as the violating service and following services will not check the deadline any more. The service being marked as the violating service will be rejected admission for some penalty period (with the same exception of minimal service rate), at which time violation flag on the service is reset to 0. The duration of the penalty period depends on by how much time the service violates the time limit.

```
do_time_sensor(max_time) {
  current_deadline = current_usage_time + max_time;
  passed_deadline = 0;
}

check_deadline() {
  if(!passed_deadline && current_usage_time > current_deadline) {
    time_violation(current_service) +=
      penalty(current_usage_time - current_deadline);
    passed_deadline = 1;
  }
}
```

The implementation of the interface for non-renewable resource management is straight-forward. Most macros simply update the pressure or progress counter in the data structure representing a resource or a principal. As an example, we give pseudo-code for `RECLAMATION_CHECKPOINT`:

```
do_reclamation_checkpoint(resid, cb, min_pres, min_prog) {
  update pressure on resid;
  if (pressure(resid) > min_pres) {
    for (each pri holding the resource) {
      usage(pri) += (time_now - last_timestamp) * held_amt(pri);
      normalized_prog(pri) = absolute_prog(pri) / usage(pri);
      update worst_pri by comparing normalized_prog counters;
    }
    /* worst_pri records the pri making the least progress */
    if (normalized_prog(worst_pri) < min_prog)
      (*cb)(worst_pri);
  }
}
```

The only trick in the code is that comparisons are made in *normalized* progress, rather than *absolute* progress, as reported directly by the application via the `PROGRESS_SENSOR` macro. The reason is that comparing absolute progress is not fair to young principals that have not yet received enough time to make progress. Intuitively, a principal holding resources for a longer period of time should have made better progress.

5.4 Compiler Support

Because code path annotations are tightly coupled with program control flow structure, we instrumented GCC and built some small tools to help users annotate their code. In general, the compiler automatically adds auxiliary annotations to complete those marked by user, and links the code annotation with the toolkit data structures. It also checks consistency of annotations and gives warning on potential discrepencies.

GCC builds a syntax tree for each function body after parsing. We added our extension to a hook between parsing and intermediate language (RTL) generation. The compiler extension traverses syntax trees to look for service admission/time sensor annotations and function exit points. When a function is marked with a service admission/time sensor annotation, the compiler inserts a call to the corresponding service exit/time sensor exit functions before each function exit.

The instrumented GCC also writes the control flow graph to a file. Our code path analyzer then reads this file and gives warnings for following cases: (1) there is a path from an entry function to a rate-sensor annotation that does not go through any service admission annotation, and (2) there are some expensive operations (e.g. loops and library function calls) enclosed by a time-sensor annotation and not enclosed by any service admission annotation.

6 Evaluation

We experimentally tested our toolkit on widely deployed software: the Flash web server, Linux kernel networking code, and NIS (yellow page) server. For each example, we annotate the code by asking ourselves the same set of questions—what services need to be separated and what resources need protection. We then tested the robustness of both the unmodified and annotated servers under various attacks. We found that both busy and claim-and-hold attack vulnerabilities exist in all test cases, and that by exploiting these vulnerabilities, an attacker could either disable, or seriously degrade the level of service. The annotated servers are much more resilient under the attacks, which demonstrates the generality and effectiveness of our toolkit. We also found situations where our toolkit has difficulty in providing protection to the desirable level. We identify some as implementation issues that can be improved by extending our toolkit, while others are fundamental limitations of our approach.

6.1 Flash Web Server

6.1.1 Annotating Flash Web Server

Flash [10] is a web server with a single-process-event-driven architecture. The main loop launches connection handlers on I/O events. We first annotate every handler function called in main loop as a service entry point. Since some of these handlers implement more than one *independent* functions—e.g., it may either read a file or execute a CGI program—we mark nested services in top-level services by functionality (e.g., `CGIStuff`). There are also some functions that contain loops or make system calls (and thus have potential to be attacked). One such example is `MakeCrossedString`, which concatenates parts of a cross-buffer string. Such functions are also marked as separate services for fault isolation. A fourth class of functions perform non-critical tasks—e.g., `ReduceCacheIfNeeded`—which we also mark as services. Altogether, 46 services are annotated.

To limit time spent in each event handler function invocation, we extract the handler function call in main loop and place it in a separate function, called `LaunchHandler`, and annotate this function with `TIME_SENSOR` .

All non-renewable resources in Flash are consumed on behalf of a connection, which is itself a non-renewable resource. Flash disables new requests when *numConnects* reaches the upper limit. The following code illustrates how we annotated function `AcceptConnections`—we insert two sensors to track usage and pressure on the connection resource. Note the pointer to the `http_conn` data structure is used as the principal identifier.

```
int AcceptConnections(int cnum, int acceptMany) {
  httpd_conn* c;
  do {
    PrepareConnOnAccept(c, newConnFD, &sin);
    numConnects++;
    RESOURCE_ACQUIRED(HTTPCONN, c, 1);
  } while (numConnects < maxConnects && acceptMany);
  if (numConnects >= maxConnects) {
    DisallowNewClients();
    RESOURCE_UNAVAILABLE(HTTPCONN);
  }
}
```

A typical HTTP connection goes through three phases: request reading and parsing, back-end processing (fetch a file from disk or execute a CGI program), and result sending. A connection makes progress when it moves to the next phase or sends out bytes. Thus, progress sensors are inserted where the "state" of a connection changes and data is sent out: `DoConnReadingBackend` and `DoSingleReadBackend` are two examples of functions with embedded progress sensors.

```
DoConnReadingBackend(httpd_conn* c, int fd, int doReqReading)
{
  switch(ProcessRequestReading(c)) {
    case PRR_DONE:
      /* end of request reading */
      PROGRESS_SENSOR(HTTPCONN, c, 10000);
      break;  /* switch connection to the next phase */
    ...
  }
}


DoSingleWriteBackend(httpd_conn* c, int fd, int testing)
{
  sz = writev(c->hc_fd, ioBufs, numIOBufs);
  ...
  /* Ok, we wrote something. */
  PROGRESS_SENSOR(HTTPCONN, c, sz);
}
```

Finally, we explicitly declare the connection resource before entering the server loop and insert a checkpoint inside the loop. The annotated main loop is shown below. `DoneWithConnection` is a Flash-provided resource deallocator, here conveniently used as the callback function for connection recycling. The choice of the parameters *min_pres* and *min_prog* are explained in Section 6.1.3.

```
void MainLoop(void) {
  RESOURCE_DECL(HTTPCONN);
  for (;;) {
    RECLAMATION_CHECKPOINT(HTTPCONN, DoneWithConnection, 5, 500);
    for (each I/O event) {
      Launchhandler(handler, tempConn, ...);
    }
    if (!newClientsDisallowed) AcceptConnections(-1, TRUE);
  }
}
```

6.1.2 Slash Attack

Flash is a very robust program: disk operations and CGI jobs are separated into helper processes rather than performed by the main process, thereby allowing the OS to protect the main process. Flash also has some built-in mechanisms to control its resource consumption; e.g. calls to `fork()` are already rate-limited. However, it is extremely difficult to write a program free of vulnerabilities, and Flash is not an exception. We found the following code in function `ExpandSymlinks`, which parses a "cold" URL that is not in server's hot URL cache:

```
/* Remove any leading slashes. */
while ( rest[0] == '/' )
  {
    (void) strcpy( rest, &(rest[1]) );
    --restlen;
  }
```

The loop has time complexity quadratic in number of leading slashes. As Flash does not limit the length of a URL, a URL with many leading slashes takes a lot time to parse: it takes 150 ms on a PIII 700 machine to remove $10,000$ leading slahes from a URL; 7 such requests per second is enough to saturate an un-annotated server.

Our attacker is a simple program that sends HTTP request "GET *//////...//id*" to the Flash server, where $id = 1, 2, 3, ...$ to avoid duplicate URLs. Under attack, the un-annotated server soon reaches the maximum number of connections. Subsequent connection requests enter a connection queue waiting to be accepted. The server will accept a connection every 150 ms. Thus server response time is greater than the connection request queue length $\times$ 150 ms.

Slash attack serves our purpose well because it shows that implementation inefficiencies that lead to DoS vulnerability may appear at unexpected locations in the source code. Ad hoc protection is not likely to cover such a vulnerability and we need a systematic approach for DoS defense. *Importantly, we knew about this problem to formulate the attack, but we did not need to have knowledge of this bug when annotating the code.*

For a Flash server that is annotated with service admissions and a time sensor with a limit of 20 ms on `LaunchHandler`, the attack has no effect on requests of hot URLs. The annotations recognizes that service `ProcessColdRequestBackend2` (see Figure 5) takes too much time on each invocation and rate limits the

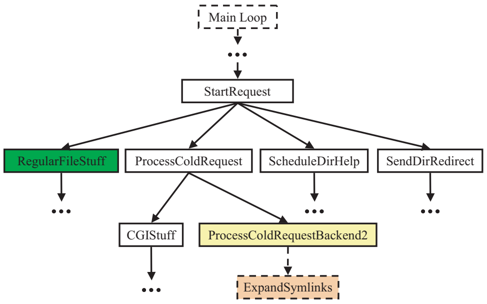


Figure 5: Position of ExpandSymlinks in Flash service hierarchy

service depending on how much time it takes for each invocation. The connection is closed on service admission rejection, so that connections do not accumulate over time. Service `ProcessColdRequestBackend2` is not invoked for "hot" URLs. By limiting CPU spent for cold URLs, we insure access to hot pages under slash attack.

no attacker	4.3 ms
attacker #slash = 0	4.3 ms
attacker #slash = 10000, original	25,000 ms
attacker #slash = 10000, annotated	5.1 ms

Table 1: Flash response time under slash attack

Table 1 compares the average response time for a "hot" 10KB file for both original Flash and annotated Flash, when the server is under slash attack. The slash attacker sends 10 requests per second to saturate the Flash server. We first measure response time to a single client without any attacker present. We then measure response time to client when there is a competing client; i.e. the attacker sends ten requests per second but with no leading slashes in the URL. The third row shows response time from an unprotected Flash server under attack, a 5000$\times$ slow down. The last row shows the response time from an annotated Flash server. The small increase of response time for annotated Flash under attack is because Flash processes a cold URL periodically and thus delays the hot request for up to 150 ms. Despite this small fluctuation, the response time from an annotated Flash server does not change by much on average under slash attack.

On the other hand, access to cold URLs is limited for annotated Flash under slash attack. The probability of success for a cold request is linear to the ratio between the user request rate and the attack request rate. For example, if an attacker sends ten requests per second (which is enough to saturate an unprotected server) and the user sends one request per second, then with probability 50% it takes no more than than $\log 0.5/\log 0.9 = 6.57$ requests to access a cold

URL. However, since nothing prevents the attacker from sending requests at a higher rate, clients may not be able to access "cold" pages in many attempts. This phenomenon shows that the effectiveness of fault isolation depends on service granularity, and sometimes depends on program classification granularity. If Flash were to further classify requests into ones with short URLs and those with long URLs, the impact of a slash attack would be further limited.

6.1.3 Slow TCP Attack

In unmodified Flash, the connection resource is recycled by an idle timer associated with each connection. The default time-out value `IDLEC_TIMELIMIT` is 500 seconds. The timer is reset by any event on the socket, such as data arrival or TCP send buffer becoming available. Thus, to launch a successful claim-and-hold attack, an attacker needs to generate an event before the 500 second timer expires. Once the available connections run out, the unmodified Flash server enters the "denial-of-service" mode, disallowing new clients. Our Slow TCP based clients can easily cause the situation to persist for days without generating very much network traffic.

By comparison, the annotated Flash server is able to recover from the "denial-of-service" mode by recycling connections. Our current toolkit implementation uses a sliding window to record pressure history. Setting *min_pres* to 5 instructs the server to reclaim resources from unproductive connections after it has been disallowing new clients for about 5 seconds. The progress of each client is tracked as follows: when a connection moves from one stage to another the absolute progress of the connection is incremented by a numerical value of 10000; when the connection is in the final result sending stage, its absolute progress increases as the bytes being successfully written. In conjunction with the *min_prog* of 500, the server enforces the following policy: a client should not stay in one stage (other than the last one) for more than 20 seconds, otherwise its normalized progress will drop below $10000/20 = 500$ and be considered "unproductive". Once in the final stage, the client should read at least 500 bytes of the server's response per second. With these resource limits, well-behaved clients including those on slow links go largely unaffected, but claim-and-hold attackers are no longer able to tie up server resources for unreasonably long periods of time.

Note that by specifying a single progress-and-pressure threshold, we may not be able to completely eliminate the vulnerability to Slow TCP attacks. Attackers can still open many connections and make each request proceed slowly while staying just above the acceptable progress threshold. To solve this problem, the programmer can specify a more refined defensive policy with the toolkit: for example, under resource pressure, at most one third of the connections can be "very slow", another one third can be "slow", while the rest have to be "fast" connections. This can be accomplished by putting more than one checkpoint with multi-level progress-and-pressure thresholds, so that the server will recycle resources more aggressively under higher pressure.

6.1.4 Overhead

Regarding programming overhead, we add in total 57 annotations into Flash source, which has more than $12,000$ lines of code. 46 of the annotations are service admission primitives that divide the program into fine-grain services. The rest are annotations on individual kinds of resources; e.g., CPU time and HTTP connections. As the annotations specify general resource policies, they should be able to defend against not only the attacks in the experiments, but also other potential attacks targeting the annotated services and resources.

In terms of request response time or server bandwidth we did not observe any performance degradation caused by annotation in our measurements. Table 2 reports the number of annotation primitives invoked on a typical HTTP request and the general cost of each annotation. The number of annotations executed varies depending on the file's size and whether it is in server cache, which affects the call graph, the number of server iterations, and the number of outgoing packets. The cost of each annotation is given in the number of instructions and "timestamp" operations. The exact cost of timestamp depends on whether the code being annotated is in kernel or user-space.

Primitives	Invocations per HTTP connection	Instructions/ timestamps per call
SERVICE Entry/Exit	$13-31$	63/2
RATE_SENSOR	n/a	25/1
TIME_SENSOR	*iterations*	36/2
RESOURCE_ACQUIRED	1	62/1
RESOURCE_RELEASED	1	42/0
PROGRESS_SENSOR	$2+pkts$	23/0
RECLAMATION_CHECKPOINT	*iterations*	121/1 per principal

Table 2: Annotation Overhead

Note the 121 instructions are the worst-case cost of `RECLAMATION_CHECKPOINT` when the pressure is high and each connection is checked. Also not shown in the table is certain background processing of the toolkit library, which executes once per second for each annotation and contains less than 20 instructions per invocation.

6.2 Linux Networking Code

6.2.1 Annotating Linux Network Code

We annotate part of Linux 2.4 network code to protect network outgoing bandwidth. Our goal is to insure that no single network activity can monopolize outgoing network bandwidth. (For incoming network bandwidth, protection on local host may not be enough, however, we may want to limit CPU time spent on incoming packets for hosts with high-bandwidth network connections.)

Initially, we mark service entry points at the "send message" function of each protocol; e.g. `udp_sendmsg`. This gives us protocol isolation. However, `icmp_reply` is an interesting case since it is called by multiple functions for sending different types of ICMP messages, e.g. `icmp_echo` and `icmp_timestamp`. To have fault isolation between different types of ICMP messages, we push the service entry at `icmp_reply` into functions for every type of ICMP message that calls `icmp_reply`. For example, `icmp_echo` is now a service entry function, while `icmp_reply` is no longer marked as a service. `icmp_send` presents another interesting case: it is called at 13 locations to report different network errors. To prevent one type of error from suppressing others, we wrap each call site as a service. In total, we mark 27 services.

Since we may not be able to get notification about delivery of packets for protocols like ICMP, we cannot apply congestion control to manage bandwidth, as the Congestion Manager does [1]. Instead, we simply rate-limit messages from all protocols except TCP.[2] On code paths that call `ip_build_xmit`, we insert a call to `ip_rate_control`, which includes `RATE_SENSOR` annotations:

```
static __inline__ int ip_rate_control(int msg_size)
{
  int res = 1;
  if (!RATE_SENSOR (sysctl_ip_max_msg_rate, 1)) {
    res = 0; ip_msg_rate_violation++;
  }
  if (!RATE_SENSOR (sysctl_ip_max_byte_rate, msg_size)) {
    res = 0; ip_byte_rate_violation++;
  }
  return res;
}
```

The user can adjust `sysctl_ip_max_msg_rate` and `sysctl_ip_max_byte_rate` through the `/proc` file system.

6.2.2 ICMP-Echo Flood Attack

To simulate ICMP-echo flood attack, the attacker sends a flood of ICMP-echo packets to the victim using the `'ping -f'` command. The attack has a 100Mbps network link and the victim is on a 10Mbps link. The victim also runs a Flash web server so that we can measure how it is affected by the attack.

Without protection, access to the Flash server on the victim machine is virtually blocked by the ICMP flood. However, the attack has almost no effect on a target system with annotated Linux code, except for the high loss rate for ICMP-echo messages.

[2] Including TCP in rate-limiting does not work because TCP will automatically back-off while other services are trying their hardest to grab bandwidth.

6.3 NIS Server

This section studies ypserv—the yellow page server available on most UNIX systems. Even though the server program itself is simple, it is interesting because it illustrates how different software architectures affect robustness. ypserv is built on top of the RPC protocol [16]. Most RPC programs are built with RPC library and tools like `rpcgen`, which handles complex tasks such as packaging a call into a message, sending it over the network, and server side message decoding. With the RPC library, the programmer only needs to provide a function that is called when a request arrives. The RPC package is valuable for constructing distributed systems, but it also comes with a potential disadvantage: its virtualization gives programmers less control on the execution of the program.

Linux ypserv-2.2 is a typical RPC server built using these tools. It starts by calling C lib functions `svcudp_create`, `svctcp_create`, `svc_register` and `svc_run`, which create transport channels, register YP services, and start a server loop that waits for requests. The main service routine `ypprog_2` is passed to `svc_register` as the callback function. `ypprog_2` dispatches incoming calls to second level routines such as `ypproc_match_2_svc` and `ypproc_all_2_svc`, and sends results back by calling C lib function `svc_sendreply`.

6.3.1 Claim-and-Hold Attacks

A client program like `ypcat` requests the entire content of a database from the server. The server handles the request by calling `ypproc_all_2_svc`. When shipping bulk data over the network, `ypserv` uses TCP as the transport protocol. We found the same vulnerability to Slow TCP attacks also exists in `ypserv`. To verify this, we built a customized version of `ypcat` that uses Slow TCP as its transport. We set up a different number of `ypcat` attackers, each requesting a database of 150K bytes. While the attack is in progress, we test the server's availability by issuing "`rpcinfo -[tu] server ypserv`" and normal `ypcat` commands from a different machine. In addition to the latest version `ypserv-2.2`, we also tested an earlier version (`ypserv-1.3`). The main difference between the two versions is that `ypserv-1.3` executes `ypproc_all_2_svc` in a forked child process, and keeps the number of children process below 40. The results are summarized in Table 3, where "Yes" means the normal client successfully gets a response from the server and "No" means the server is unable to reply.

The results show that ypserv-2.2 becomes unresponsive under the presence of *any* slow `ypcat` attackers. This is not surprising since it is an iterative server that handles only one call at a time. Interestingly, version 1.3 with concurrency support also failed with just 1 slow sender, and damage was done to not only

	ypserv-2.2		ypserv-1.3	
	`rpcinfo`	`ypcat`	`rpcinfo`	`ypcat`
1 slow sender	No	No	No	No
1 slow reader	No	No	Yes	Yes
40 slow readers	No	No	Yes	No

Table 3: Server Availability under Slow ypcat attacks

TCP but UDP services as well. The reason is that `svc_run` essentially implements a *poll* loop as in Flash, but using synchronous I/O. When data arrives on a registered channel, the RPC library tries to decode the request message. If the request message is sent slowly, the main server process blocks on a `read` system call until the entire message arrives. During this time, the server is unable to reply to new requests. The concurrency, however, does help the server survive slow reader attacks, as they are handled by children processes. When the number of slow readers reaches the limit, `ypcat` starts to fail, but the main process continues to respond to `rpcinfo` and other YP clients such as `ypmatch`.

We found that merely annotating ypserv does not give us resilience to Slow TCP attack because the activities we would like to monitor actually occur inside the RPC library rather than the application. Therefore, we really need to annotate the RPC library. However, the effectiveness of doing so is hampered by the library's use of synchronous I/O. We suggest that a more robust RPC library implementation should employ the architecture of the Flash web server, in which (1) low-level stub functions are processed in non-blocking handlers, and (2) user applications like `ypserv` are invoked as helper processes. If these changes were made, our annotation toolkit would effectively protect the RPC library.

6.3.2 Busy Attacks

There is an easy way to busy attack a ypserv-2.2 NIS server when there is a big database: simply invoke many "`ypcat <big database>`" simultaneously to ask the service to send the whole database over network. For a database of size 1.7MB, it takes about 20 ms for server to complete the transmission, during which the server does not process any other requests because of RPC's mutual exclusion property. Attacking a NIS server with `ypcat` flood virtually blocks all NIS operations using TCP, e.g. `rpcinfo`. Operations that use UDP still go through because they are in a different queue than TCP in `select()`.

We annotated the NIS server by wrapping each NIS operation as a service so that `YP_ALL` requests (sent by `ypcat`) will not consume all the resources. An annotated NIS server continues to respond to other YP requests under a `ypcat` attack, except access to `YP_ALL` is very slow. However, this is not satisfactory because `YP_ALL` access to database *group* is required for each log-in. Since *group* is usually a very small database, it is not vulnerable to a `ypcat` attack. Generally, we do not want to let `ypcat` attacks on large databases affect access to small databases. Since there are usually only a small number of databases on a NIS server, we can solve this problem by associating a "dynamic" service for each type of operation on each database, so that "`ypcat group`" and "`ypcat passwd`" belong to separate services. To support dynamic service, we need to add one new primitive, `DYN_SERVICE_ADMISSION(svc_id, min_rate)`, which is same as `SERVICE_ADMISSION` except it takes an extra parameter *svc_id* for service id.

7 Limitations

Our approach has several limitations. First, in many cases our approach limits only the scope of damage because it cannot distinguish between "good" and "bad" requests that happen to follow the same code path. In other words, annotations simply augment the classification mechanisms already embedded in the code; they do not add any new ones of their own. To further differentiate between "good" requests and "bad" requests, additional classification mechanisms must be added to the program so that these requests effectively follow different code paths. For cases where separating services according code paths is not fine-grain enough, as we saw in the experiment on the NIS server, we believe that adding `DYN_SERVICE_ADMISSION` to the toolkit will be necessary. We are currently extending the toolkit to support such a facility.

Second, the current toolkit is only applicable within a single process because the sensors and actuators need to share state, and thus, they work only within a single memory space. This means our toolkit will not work with the current implementation of Apache, for example. It is not clear that an IPC facility can help extend the mechanism to multi-process programs because IPC overhead will likely hinder fine-grain protection. However, for multi-process programs, it is also possible to apply the protection separately for each process. We need more experience to say how effective that will be.

Third, rate-limiting controls only the quantity of resources consumed by each service, but not the order that resources are consumed. Sometimes it is desirable to change the order that we allocate resources, especially when some resource consumers are latency-sensitive. For example, in addition to specifying a rate for all non-TCP packets, we may want to bump TCP packets to the front of the transmission queue. Not being able to schedule resource sometimes forces the user to be more conservative in specifying resource limits. To be able to schedule resource allocation would require support for concurrency within a process, so that the program execution can save the state of the current service task and switch to another service.

8 Conclusions

This paper presents defensive programming as a new approach to offer proactive DoS attack protection. After first identifying two basic types of DoS attacks—busy and claim-and-hold—we build a toolkit that provides an interface programmers use to annotate their code. With compiler assistance, annotations are translated into runtime sensors and actuators that watch for resource abuse and take the appropriate action should abuse be detected. The main strengths of this approach are that it offers fine-grained intra-process protection, can be systematically applied to existing code, protects software from unknown attacks, and puts a minimal burden on the programmer.

Like any mechanism, however, the effectiveness of our approach depends on whether a good defensive policy can be specified, which is the responsibility of the programmer. Our experience with DoS attacks and applications has greatly influenced the design of the annotation interface in order to accommodate the most common policies, but the interface is by no means complete. Also, even with the help of our toolkit, non-trivial programming effort is still required: (1) programmers need to mark service entry points and identify where their programs acquire/release/consume resources, and (2) system administrators need to set system-dependent parameters (e.g., rate limits). Our view is that just as programmers are responsible for making their programs correct, they should also be responsible for making them defensive; we merely provide a set of tools to help simplify this task. Preliminary experience suggests that the programming burden is modest, but we expect to extend and refine the tools as we gain more experience.

Acknowledgments

We would like to thank the anonymous reviewers and Greg Ganger, our shepherd, for helping us improve the clarity and focus of the paper. This work was supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, and Intel Corporation.

9 REFERENCES

[1] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *Proceedings of the Fourth USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 2000.

[2] Apache Software Foundation. Apache Web Server. http://www.apache.org/.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 1999.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth USENIX Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, May 1996.

[7] K. Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. *Master Thesis, MIT*, June 1999.

[8] Linux NIS(YP) Server. http://www.linux-nis.org/nis/.

[9] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, October 1996.

[10] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX '99 Annual Technical Conference*, June 1999.

[11] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 134–145, December 1993.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[13] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a Denial of Service Attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Computer Security and Privacy*, May 1997.

[14] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[15] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 1999.

[16] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. *Request for Comments (RFC) 1831*, August 1995.

[17] C. Villamizar, R. Chandra, and R. Govindan. BGP Route Flap Damping. *Request for Comments (RFC) 2439*, November 1998.

[18] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Computer Security and Privacy*, May 2001.

[19] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.