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

Xiaohu Qie, Ruoming Pang, and Larry Peterson
Princeton University

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. Experience with three widely-deployed network services demonstrates the effectiveness of the toolkit.

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. For example, setting up router filters to disable directed broadcasts and blocking outgoing packets with spoofed source IP address [5] can effectively shut down the *smurf* attack. To address the problem that a widely deployed BGP implementation tends to fail due to high routing update volume [16], a route damping feature has been added to BGP [10] implementations.

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. In other words, after the damage is done.

Instead of reacting to attacks after the fact, a better solution is to make the software itself *defensive*, by which we mean the programmer embeds *general* protection mechanisms into the software that offers *systematic* and *proactive* protection against DoS attacks. Ideal defensive software will guarantee availability even under a previously unknown DoS attack.

Towards this goal, this paper describes our experience developing mechanisms to help programmers systematically build robust software. The key idea is to insert annotations into code that monitor and control the execution of the program at runtime. These annotations serve both as sensors that detect the anomaly, and actuators that change the control flow of a program when it detects that defense 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 is possible to confine the damage of an attack without negatively affecting other aspects of the program.

We have developed a toolkit consisting of an annotation interface, a runtime library, and a set of compiler extensions. A programmer uses the interface to annotate code as a means of specifying a resource management policy. Each annotation gives a hint regarding the code path it is embedded in. The compiler extension sets up the relationships between annotations by analyzing the control flow graph of the program, and generates necessary code to be executed at annotated points. At runtime, appropriate monitor and control functions are invoked as control flow passes through these annotations.

One important aspect of the annotation toolkit is that it is designed to thwart common DoS attack characteristics. Therefore, when using it, programmers do NOT have to scan their code for a specific implementation vulnerability and then fix it, as they do when writing a software patch. Instead, they just follow some general guidelines and reason about the resources used by the program. We believe this feature significantly facilitates proactive and systematic protection. In addition, annotating is less intrusive than rewriting code, making it especially valuable to protecting legacy software.

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, this paper identifies a class of attacks that exploits a vulner-

ability 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 TCP/IP protocol stack, the Flash [9] web server, and the Linux NIS servers. 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. For example, 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, 13].

Our approach has the flavor of anomaly detection, but with a focus on resource usage rather than security. Because 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.

One obvious question is why not just do profiling? We think our toolkit is a more general solution. First, profiling does not cover all 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. To offer protection, it is necessary to know what the resource is, as well as where and how it is being used. Our annotation interface allows a program to provide such information. Second, 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 defense mechanism built into the program, the only action left available is to kill the victim process, which is a DoS attack in its own right. Third, 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. However, to infer the behavior distribution from profiling data is a hard problem that does not have a good answer for the general case.

2.2 OS Mechanisms

There has been an ongoing effort in the OS community to build new mechanisms and specialized OS to provide service differentiation and guarantees. For example, Resource Containers [2], are an abstraction that takes over the process' role as the primary resource principal. In order to receive a CPU share, a process must bind to a container associated with a certain activity. This mechanism 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, thus achieving flexible resource management through an additional level of indirection. The Scout operating system [8, 14] 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.

Our API differs from the one of resource container's in how the defensive policy is specified in the software. The resource container API requires a clear identification of the resource principal or some knowledge about the attack in order to offer protection. For example, to defend against TCP SYN flooding attacks, the programmer needs to know that "TCP SYN packets from a particular subnet are sent by attackers" so as to bind all connections from that subnet to a container with a numeric priority of zero [2]. In contrast, the design of our annotation API guides user in identifying resource principals. In particular, we observe that resource principals should be program functionalities, rather than clients, when there are too many or anonymous clients.

Another difference between resource containers and our toolkit is the granularity of protection. An important contribution of resource containers is the separation of resource principals and execution domains, but it does not change the fact that process (or thread) is still the execution domain, and resource management policies are ultimately effected via process scheduling *between* different 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. For these applications, *intra-process* protection is more important, since we do not want to penalize the entire process when just one of the functions is supports is being abused. We believe monitoring and controlling code paths is a natural, yet efficient way to provide intra-process resource management. Using annotations, we can effectively isolate code paths corresponding to distinct functionalities, and thus offer fine-grained intra-process protection.

A third difference is that our protection is not limited to CPU scheduling, as we observe that non-renewable resources cannot be protected by scheduling; they must be recycled.

Finally, annotating code is more programmer-friendly

than imposing a new OS architecture or abstraction, which often requires re-architecting code. This is especially true with Scout, which prescribes a particular structure for implementing services.

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 done by Engler *et al.* [3, 4] 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 software 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 pointer 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 resource 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 [11] seems to support this point of view. Our approach has the flavor of static analysis, but the main difference is that we check for possible "rule" violations at runtime, with a focus on resource usage.

3 DoS Attack Characterization

Researchers have studied may DoS attacks [12, 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, which 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 a large volume of requests at it, suggesting that rate limiting and load conditioning [17] would be an effective defense. However, DoS attacks can also be carried out in a way that renders rate limiting strategies ineffective. The example in the following section illustrates this possibility.

3.1 Slow TCP

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 byte stream 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 receive the data very slowly, simply by not opening the advertise window. The server side TCP would interpret the closed 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 slowing 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 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 "`UserAgent: 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 the server's output. The goal of the attacker is to keep the connection alive as long as possi-

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

ble. 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 and Flash [9]. 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. 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.

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 *nonrenewable 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 nonrenewable, the attacker tries to acquire as many resource as possible, and then not release them. This form of attack does not require flooding to make the target busy, in fact, the server is basically idle.

In the rest of the paper we denote an attack targeting a renewable resource a *busy* attack, and an attack targeting a nonrenewable resource an *claim-and-hold* attack. although 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 nonrenewable 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 nonrenewable resource.

These "exceptions" are not special cases, but in fact, a general 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 nonrenewable 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 Defense Strategies

Our strategy is to divide a program into *services* and balance resource usage among services, thereby confining the impact of an attack to the individual service being attacked. We consider both renewable and non-renewable resources, in turn.

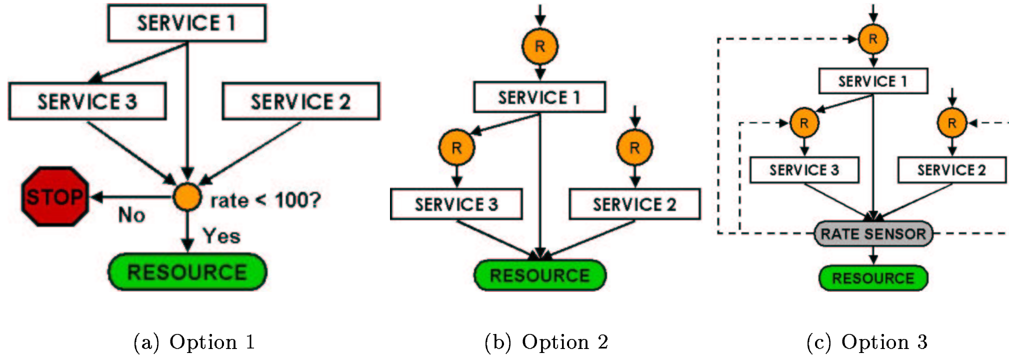
4.1 Busy Attack Defense

4.1.1 Services and Code Paths

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, each ICMP service is handled by a distinct function with name `icmp_<service>` (e.g. `icmp_echo`). We would like to divide a program into services according to code paths, but it is unclear how to have the compiler automatically do this. Therefore, 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.

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 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. Also, the service hierarchy structure is clearly represented by the function call graph.

Services can be disjoint or nested. For example, TCP-send and UDP-send are disjoint services, while the service of IP option 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



(a) Option 1 (b) Option 2 (c) Option 3

Figure 1: Rate Limit

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 vulnerable, all other IP options can be still be handled normally.

Since 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 program execution paths based on resource usage as intra-process "scheduling" among services. However, it is hard to precisely tell how expensive code paths are, and we do not know which code path will be attacked, so there are two interesting questions in rate-limiting code paths: (1) where to place rate controllers that change execution paths; and (2) how to decide the rate limit, or more generally, how to decide whether or not to switch out of current code path each time execution reaches the rate controller.

4.1.2 Placement of Rate Controllers

Placing rate controllers in an ad hoc way may leave holes to be exploited. Putting them everywhere (e.g., at every branch point) also does not work. First, rate limit violations need to be handled in program-specific ways, so it takes programmer effort to write handlers for rate controllers. Placing rate controllers everywhere places an increased burden on program writers. Second, placing rate controllers everywhere incurs too much execution time overhead, especially when they are placed on frequently executed paths.

Since the goal is to control resource usage, the first option we considered is to put rate controller before resource consumption, as shown in Figure 1(a). However, this approach does not balance resource usage among the sharing services. Furthermore, aborting the service operation at that particular location may not be easy. To address both problems, we argue that *service entry points are the right place to put rate controllers* (Figure 1(b)). Since a service is the unit of fault isolation, activities within the same service share fate, and therefore it is better to not begin processing a service request if it cannot acquire enough resources to finish. Also, it is usually easier to abort or delay processing a service request at the entry point than in the midst of processing.

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, we observe that 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. To achieve better precision with our approach, the programmer must define finer-grained services.

4.1.3 Computing the Rate Limits

With rate controllers at service entries points, we need to decide whether to admit a service or to reject it. Here, we may let user specify a maximal rate for each service—this is the most common approach used in existing systems. However, as we do not know where within the code the actual vulnerabilities are, it requires much effort and experience to set an appropriate limit for each service, and there is a risk of being either too conservative or too optimistic. Furthermore, the choice is often host-specific and cannot be easily shared or reused. The reason behind this difficulty is that it is unclear how service rate limits map onto actual resource usage.

Although neither option is satisfactory, they gave us some insight into how to rate control code paths and let us realize that a rate controller actually needs to be separated into two parts: an *admission controller* at service entry, and a *rate sensor* where the resource is consumed.

We ask the user to specify where the program consumes resources and set an overall rate limit on each resource. Based on feed-back of resource usage, we rate control the services so that resources are utilized near the rate limit, while giving each service a fair share (shown

in Figure 1(c)). Further details on how to compute each service's share are presented in the next section.

4.1.4 Controlling Continuous Resource

The discussion to this point assumes that resources are always consumed at particular locations of the program. However, we distinguish between two types of renewable resources: *discrete*, which includes almost all renewable resource except CPU time (e.g. network/disk bandwidth); and *continuous*, which includes CPU time. Unlike discrete resources, CPU time is spent continuously as the program executes, and so needs to be managed differently.

Consider a simple scenario in which we want to limit the time spent in a particular function to less than 10% of the total time spent executing the program. The straightforward time control may not give us what we really want because (1) the function may be invoked by more than one services and we do not want any of them to monopolize the time quota for the function by invoking the function a lot of times; and (2) the function may invoke several independent services, and if one of them takes a lot of execution time on each invocation, other services will get less time.

For the first case, we consider the function to be a discrete resource that is shared by parent services, and apply the rate control method described above. The second case can also be handled by balancing time usage among services invoked by the function, though it is trickier as services can be nested. *However, the real difficulty lies in distinguishing between the two cases—whether the function is invoked too frequently, or the function takes too much time to execute on each invocation.* In order to make this distinction, we need program-specific knowledge of the normal rate that the function should be invoked and how much time the function should take on each "normal" invocation. The former can be specified as a discrete resource rate limit, and the later can be given with a time limit on each invocation. With the time limit, we use a simple heuristic to find the service in which we spend too much time: it picks the service inside which the execution time passes the deadline. As a heuristic, this approach may not be the best method, but it is easy to understand and implement, and works well in practice.

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, as all we need to control is the execution frequency and duration of different code paths. Protecting nonrenewable resources, however, is a different story. Attackers holding the resource do not necessarily have to remain active once the resource is acquired.

Protecting nonrenewable 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 what metrics should be used to decide what resources to reclaim and when such operations should be performed.

4.2.1 Metrics

Consider the Slow TCP attack against web servers; the resource in question is the server connection. In both the Apache and Flash implementations, there is no explicit replacement policy. The connection resource is returned when the client request is completed. If the connection table is full, the server simply rejects new requests. However, both of these systems have defense mechanisms to address this problem. For example, Flash has 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. One 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 is highly dependent 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, and it should 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 use a metric *pressure* 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.

For a particular resource that needs protection in a specific program, programmers can develop other metrics that fit better into the situation. As a general toolkit, we currently only support interfaces to keep track of progress and pressure.

4.2.2 Placing Sensors and Controllers

There are often well-defined points in the program where nonrenewable resources are accessed. If resource allocators and deallocators are defined, we can place annotations inside them to track the principals that hold the resource. This is also the place to watch for pressure: an unsuccessful allocator call is a sign that the resource has become scarce. Some abstract nonrenewable resources are not accessed via an explicit function interface, in which case we need the programmer to specify where the resource is acquired and released.

Progress is typically tracked in two ways. If the principal in question generates output of some kind, the unit of the output is a natural progress metric; e.g., how many bytes read/written by a server process, how many packets forwarded on a tunnel, etc. Progress sensors should be placed where the output is generated. Under another scenario, an entire task can be broken into stages, where progress is made 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 hot file vs. a cold file). 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.

When an event causes the pressure to exceed a tolerable level, we may need to control resource usage by performing a reclamation. We also need to examine the pressure periodically, as it could build up even in the absence of activity. This implies that we need to insert a resource checkpoint that is periodically visited by the control flow. For most server programs this is not a problem as they are iterative by nature. An important issue, however is that when an action is taken, is 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.

Additionally, when placing a resource controller 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-controller, 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 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 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 the entry point for a service, 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_CONTROL (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_CONTROL (sysctl_icmp_max_msg_rate, 1))
  icmp_msg_rate_violation++;
if (!RATE_CONTROL (sysctl_icmp_max_byte_rate, msg_size))
  icmp_byte_rate_violation++;
ip_build_xmit(...);
```

`RATE_CONTROL` 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_CONTROL (max_time)
This annotation is used to control the execution time of a code path 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_CONTROL` so that admission to services invoked by event handlers will be bounded by the time limit.

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

5.2 Nonrenewable Resource Management

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

RESOURCE_DECL(resid)
This annotation declares a nonrenewable 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(resid, amt)
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(resid, p, amt)
This annotation updates the progress metric of a principal. The use of the opaque pointer "*p*" should be consistent with that in `RESOURCE_ACQUIRED` and `RESOURCE_RELEASED`.

RESOURCE_CHECKPOINT(resid, callback, min_pressure, min_progress)
This annotation is the resource controller that performs recycling. By default, it takes resources back from the principal making the least progress. Programmers can configure the operation with two additional parameters: *min_pressure* specifies that actions should be taken only when the pressure exceeds certain threshold; *min_progress* 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 that is invoked by the controller. 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 controller, time controller, 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 control 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 will insert 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_time_limit();
  set current_service to svc_id;
  if (service within min_rate || there is no violation)
    return 1;
  return 0;
}

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

The rate controller structure contains a rate counter *for each service* that uses the rate controller, 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-control 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 controller. Below is the pseudo-code for rate controller:

```
do_rate_control(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_CONTROL annotation includes the current function and all its subroutines. At entry TIME_CONTROL computes and stores a deadline in global variable *current_deadline*. When TIME_CONTROL 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_CONTROL uses wall time.) Within the scope of time-control, 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_control(max_time) {
  current_deadline = current_usage_time + max_time;
  passed_deadline = 0;
}

check_time_limit() {
  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 nonrenewable 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 RESOURCE_CHECKPOINT:

```
resource_checkpoint(resid, callback, min_pressure, min_progress)
{
  update_pressure(resid);
  if (pressure(resid) > min_pressure) {
    for (each pri holding the resource) {
      update_age(pri);
      usage(pri) += (time_now - last_timestamp) * held_amt(pri);
      norm_progress(pri) = abs_progress(pri) / usage(pri);
      if (norm_progress(pri) < worst_progress) {
        worst_progress = norm_progress(pri);
        worst_pri = pri;
      }
    }
    if (worst_progress < min_progress)
      (*callback)(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 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 suspicious discrepencies. We also built a tool to identify potential program vulnerabilities, but it has not been very successful as currently it gives too many false positives.

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/time control annotations and function exit points. When a function is marked with a service/time control annotation, the compiler inserts a call to the corresponding service/time control 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-control annotation that does not go through any service admission annotation, and (2) there are some expensive operations enclosed by a time-control 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 [9] is a web server with a single-process-event-driven architecture. The main loop launches connection handlers on I/O events. We first annotated 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, we annotated 46 services.

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_CONTROL` .

All nonrenewable resources in Flash are consumed on behalf of a connection, which is itself a nonrenewable 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:
      PROGRESS(HTTPCONN, c, 10000); /* end of request reading */
      break;  /* switch connection to the next phase */
    ...
  }
}

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

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_pressure* and *min_progress* are explained in Section 6.1.2.

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

6.1.2 DoS Vulnerabilities and Defense

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 bug-free program, 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 to 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 150ms 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 $id = 1, 2, 3, \ldots$ to avoid duplicate URLs. Under attack, the un-annotated server soon reaches maximal number of connections. Following connection requests enter a connection queue waiting to be accepted. The server will accept a connection for every 150ms. Thus server response time is greater than connection request queue length $\times 150ms$.

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

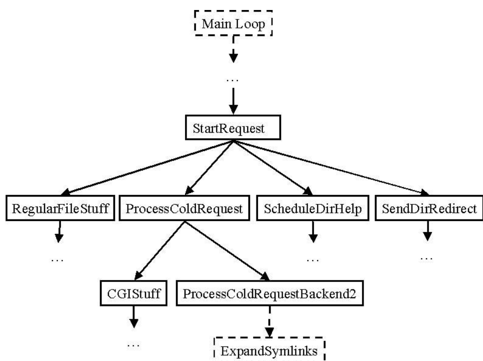


Figure 2: Position of ExpandSymlinks in Flash service hierarchy

For a Flash server that is already annotated with service marks and a time control (of 20 ms) on `LaunchHandler`, the attack has no effect on requests of hot URLs. The annotations recognizes that service `ProcessColdRequestBackend2` (see Figure 2) takes too much time on each invocation and rate limits the 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 measured response time to a single client without any attacker present. We then measured response time to client when there is a competing client; i.e. the attacker sends ten requests per second but with zero leading slash in URL. The third row shows response time from an unprotected Flash server under attack, a 5000× 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 150ms. 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.

Slow TCP Attack

In unmodified Flash, the connection resource is recycled by an idle timer associated with each connection. The default time-out value `CGI_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 only 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_pressure* 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_progress* 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 a slow link 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 checkpoints with multi-level progress-and-pressure thresholds, so that the server will recycle resources more aggressively under higher pressure.

6.1.3 Overhead

Regarding programming overhead, we added in total 57 annotations into Flash source, which has more than 12,000 lines of code. In terms of request response time or server bandwidth we did not observe noticeable performance degradation caused by annotation. 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_CONTROL	n/a	25/1
TIME_CONTROL	*iterations*	36/2
RESOURCE_ACQUIRED	1	62/1
RESOURCE_RELEASED	1	42/0
PROGRESS	$2 + pkts$	23/0
RESOURCE_CHECKPOINT	*iterations*	121/1 per principal

Table 2: Annotation Overhead

Note the 121 instructions is the worst-case cost of `RESOURCE_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 annotated 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 marked service entries 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 entry. `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 marked 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 is defined as follows:

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

[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.

'ping -f' command. The attack has a 100Mb network link and the victim is on a 10Mb 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 Flash server on victim machine was virtually blocked by the ICMP flood. However, the attack has almost no effect on an annotated victim, except the high loss rate for ICMP-echo messages.

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 [15]. 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.

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 got a response from the server and "No" means the server was unable to reply.

The results show that ypserv-2.2 become unresponsive

	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

under the presence of *any* slow ypcat attackers. This is not surprising considering that it is an iterative server handling 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 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 did help the server survive slow reader attacks, as they are handled by children processes. When the number of slow readers reaches the limit, ypcat started to fail, but the main process continued 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.

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 20ms for server to complete the transmission, during which the server does not process any other request 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 will 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 ypcat attack, except that access to YP_ALL is very slow. However, this is not satisfactory because

YP_ALL access to database *group* is required for each login. Since *group* is usually a very small database, it is not vulnerable to 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. Thus ypcat group and ypcat passwd will belong to separate services. To support dynamic service, we need only one new primitive: DYN_SERVICE_ADMISSION(svc_id, min_rate) which is same as SERVICE_ADMISSION except that it takes an extra parameter *svc_id* for service id.

7 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 is 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. Hopefully, as we gain more experience with attacks and different systems, we will be able to further extend and refine the API.

Another limitation of the approach is there may be situations that require resource scheduling within a process. With our toolkit, the user defines resource overload conditions with a limit on each resource, the assumption being that there exists some limit under which resource consumption is always fine. However, sometimes it is desirable to be able to change the order that we allocate resources to services. For example, in addition to specifying a rate limit 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 resources we need support for concurrency within a process, so that the program execution can save the state of the current task and switch to another one.

Another interesting direction to explore is adding interaction between user and kernel space. We can imagine putting resource sensors inside the kernel to achieve more precise control on kernel resource consumption, while letting actuators take actions in user-space programs to achieve intra-process resource management.

8 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] 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.

[3] 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.

[4] 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.

[5] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. *Request for Comments (RFC) 2267*, January 1998.

[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] 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.

[9] 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.

[10] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). *Request for Comments (RFC) 1771*, March 1995.

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] 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.