Preventing Denial-of-Service Attacks on a μ-Kernel for WebOSes

Jochen Liedtke Nayeem Islam Trent Jaeger
IBM T. J. Watson Research Center
jochen@watson.ibm.com

Abstract

A goal of World-wide Web operating systems (WebOSes) is to enable clients to download executable content from servers connected to the World-wide Web (WWW). This will make applications more easily available to clients, but some of these applications may be malicious. Thus, a WebOS must be able to control the downloaded content's behavior. In this paper, we examine a specific type of malicious activity: denial-of-service attacks using legal system operations. A denial-of-service attack occurs when an attacker prevents other users from performing their authorized operations even when the attacker may not be able to perform such operations. Current systems either do little to prevent denial-of-service attacks or have a limited scope of prevention of such attacks. For a WebOS, however, the ability to prevent denial-of-service should be an integral part of the system. We are developing a WebOS using the L4 μ-kernel as its substrate. In this paper, we evaluate L4 as a basis of a system that can prevent denial-of-service attacks. In particular, we identify the μ-kernel-related resources are subject to denial-of-service attacks and define μ-kernel mechanisms to defend against such attacks. Our analysis demonstrates that system resource utilization can be managed by trusted, user-level servers to prevent denial-of-service attacks on such resources.

1 Motivations

One of the results of the explosive growth of the World-wide Web (web) is the popularization of downloading executable content from the remote sites. However, downloading content from an arbitrary site increases the client's risk of executing malicious code. For example, clients can easily download Java applets using their favorite web browser. Although mechanisms for controlling the access rights of Java applets exist, the ability of these mechanisms to prevent *denial-of-service attacks* is limited. For example, FlexxGuard[1] limits the number of windows that can be opened, processes that can be triggered, etc. for a Java applet. However, FlexxGuard can control only individual Java applets. It cannot take remedial actions should the combination of applets executing result in a denial-of-service.

Denial-of-service attacks is characterized by an attacker preventing "an authorized user from referring to or modifying information, even though the [attacker] may not be able to refer to or modify the information." [6]. This definition encompasses some types of denial-of-service attacks, which we call *security malfunctions*, in which an attacker implements an attack by changing the operations of the system. In our analysis, we focus on denial-of-service attacks that use the system's operations in an abusive manner. An example of such an attack is an attacker writing data until the file system is full.

In this paper, we describe the use of μ-kernels to control processes to prevent denial-of-services attacks on WebOSes. Concrete security policies and architectures vary between operating systems, applications and domains. μ-kernel security does not imply a specific security policy but concentrates on security safety and security support.

- *Security safety:* To what extent can μ-kernel mechanisms be used for security attacks (or are themselves insecure)? Can these attacks be controlled?

- *Security support:* Are the μ-kernel mechanisms sufficient to implement any (computable) security policy?

In this short paper, we examine the use of the L4 μ-kernel[4] as a substrate for building WebOSes that can prevent denial-of-service attacks. L4 is designed to provide a small number of key operating system services (e.g., processes, address spaces, threads, IPC, etc.) in an efficient manner. Additional system functionality is implemented by servers that execute in user-space. An L4-based WebOS clearly depends on the functionality L4 provides to prevent denial-of-service attacks. We show that L4 enables:

- system resources to be managed by user-level servers (section refresources) and that
- servers to be constructed in such a way that they cannot be blocked by denial-of-service attacks (see section 4).

1.1 Informal Definitions

We use the term *security policy* to describe a set of rules which are enforced by the *security system*: the security system implements the security policy, and the security policy ensures that the above mentioned security properties hold.

1. *Trusted computing base* (TCB) is a set of assertions describing the trusted behavior of all components. A behavior is trusted, if the security system relies on its correct behavior. The correctness of the TCB is a *conditio sine qua non* for the security system. Usually, violating an assertion of the TCB penetrates the security system. For security and robustness reasons, as few components and as few assertions as possible are classified to be trusted. Typical examples for a WebOS include the hardware processor, memory, μ-kernel, basic servers and some drivers.

   Untrusted components have no assertions in the TCB. For a partially trusted component, the TCB specifies only some aspects of its behavior. The functionality of completely trusted components is defined completely by the TCB.

Our notion of a TCB as a set of assertions differs from the more common notion of a TCB as a set of trusted components because the component definition is to course grained for hardware and device drivers. Any piece of hardware has to be trusted. For example, the hardware is trusted not to short circuit, not to block the bus, and not to modify other bus signals. On the other hand, depending on system requirements, the functionality of some the devices might be irrelevant for the security system. The requirements for device drivers are similar: any device driver that is part of TCB must not disables interrupts longer than 20 μs, although it might be irrelevant whether the loudspeakers are driven correctly. Defining the TCB as a set of assertions permits us to model these partially trusted components.

2. A *security malfunction* is any discrepancy of a completely or partially trusted component from its trusted behavior. The most noteworthy examples are hardware defects and programming bugs.

3. A *security attack* is any attempt to penetrate the security policy provided the attempt is not based on a security malfunction. (We do not differentiate between a planned and purposeful attack and an accidental behavior having similar characteristics. The security system has to neutralize both in the same way.)

Assume that a trusted (but incorrect) driver disables interrupts forever. This is considered to be a malfunction, not an attack. However, if someone tries to modify the driver without proper authorization or to replace it by an untrusted driver, we have a security attack. When a trusted disk driver no longer works due to a bug or a hardware defect, this is regarded as a malfunction. When the network tries to deliver packets with 10 MHz, this should be treated as an attack.

However, programming bugs or hardware malfunction *are* attacks if they occur in untrusted components or if they relate only to the untrusted behavior of a partially-trusted component.

The difference between malfunctions and attacks is important, since the security system *can* only defend against attacks. Dealing with malfunctions (fault tolerance) is a completely different job, also depending on specific hardware support.

4. A *confidentiality attack* aims to read data which should not be readable due to the security policy. Such an attack is particularly dangerous if it is directed against authentication data, like pathwords or secret keys.

5. An *integrity attack* tries to modify data and/or code unauthorized. A successful attack of this type can change the effective semantics of the entire system.

6. A *denial-of-service* attack tries to reduce or even deny a service which is guaranteed by the security policy.

This paper deals solely with denial-of-service attacks. Other attacks and security malfunctions are not considered.

1.2 Direct Attacks

Direct denial-of-service attacks either try to overload or to monopolize a resource. If they succeed, the system no longer works properly or even starves completely. Overloading attacks are for example generating bulk mail or starting a large amount of space and/or time consuming jobs. Monopolizing attacks become possible, if a task can reserve a resource exclusively, for example a diskette drive or a special memory region.

A general strategy against monopolizing attacks:

a. Permit exclusive reservation only if the resource cannot be multiplexed.

b. Depending on the criticalness of a resource, permit exclusive reservation only for trusted processes.

c. There should be a supervising instance which can release the resource in question. Ensure that the supervising instance does not depend directly or indirectly on the resource.

Finding strategies against a overloading attack is harder, since differentiating between legitimate high load and an overloading attack is resource-dependent and system-specific. In most cases, potential attackers are therefore not killed but their activity is logged and their resource utilization is limited.

[1]

General strategy:

a. Allocate and enforce resource budgets for subjects.

b. Monitor resource usage.

c. Always provide a supervising instance which can reduce the current budgets and potentially release the resource (if it is locked). Ensure that it does not depend directly or indirectly on the resource being supervised.

[1]How can we recognize a potential attacker? This is impossible. What if we confuse a legitimate (paying) customer for an attacker?

1.3 Indirect Attacks

Instead of attacking the resource, the manager controlling and supervising the resource might be attacked. There are three types of manager attacks:

- *Manager-overloading attack.*
  Generate junk requests such that the manager can no longer operate "fast enough" for legal requestors leading to a degradation of service.

- *Manager-resource attack.*
  If the manager itself relies on a secondary resource, a denial-of-service attack to the secondary resource blocks the manager. Processor time is a typical example for such a secondary resource.

- *Manager-integrity attack.*
  Modify the manager so that it no longer works properly.

We can regard the manager itself as a resource. Then the first two mentioned attacks can be dealt with as direct attacks. The third indirect attack is actually an integrity attack and will therefore not be discussed here since We assume that integrity attacks prevented by the security system.

1.4 The L4 μ-Kernel

This paper discusses denial-of-service attacks on μ-kernels. The μ-kernel we use as our example is L4 [4, 5, 2]. We assume the reader has some familiarity with the L4 μ-kernel interfaces and with small kernels design issues in general.

The analysis that we present could have been implemented on a kernel such as the exo-kernel or μ-choices. What we illustrate is a basic technique for providing assurance that a μ-kernel is free from denial-of-service attacks.

2 Denial of service attacks on μ-kernels

To the μ-kernel resistant against denial-of-service attacks we must show that

- all resources can be managed by servers (section refresources) and that
- servers can be constructed in such a way that they cannot be blocked by denial-of-service attacks (see section 4).

3 μ-Kernel Resources

Memory, I/O ports, interrupts, threads, tasks, processors and *time* are the resources over which the μ-kernel exerts some influence. Since a μ-kernel implements mechanism and not policies, resources should as far as possible be maintained and allocated by servers. Denial-of-service attacks could then be handled by the according servers. This permits to implement flexible and system-specific strategies.

For making the μ-kernel resistant against denial-of-service attacks we must therefore show that (a) all resources can be managed by servers and that (b) servers can be constructed in such a way that they cannot be blocked by denial-of-service attacks. Topic (b) is discussed in section 4.

3.1 Memory

Memory is maintained by so-called pagers. The root server of this type is σ_0, the pager which initially owns all available memory and io ports. By means of mapping and granting operations, higher-level pagers are implemented on top of σ_0.

σ_0 is part of the trusted computing base. All components, including the μ-kernel, rely for example on the property that the memory they get from σ_0 exists physically and is not aliased with other physical memory.

After initialization, one could imagine an attack based on mapping. The attacker (see figure 1) generates an auxiliary address space (a new task) and maps one of its own pages multiply into the auxiliary space. This attack does not consume user memory but page tables. For mapping the data page to 512 virtual addresses, up to 2 Mbyte of page tables might be required. Without an appropriate control mechanism, the sketched attack would consume all page table space so that all further mapping operations would be denied.

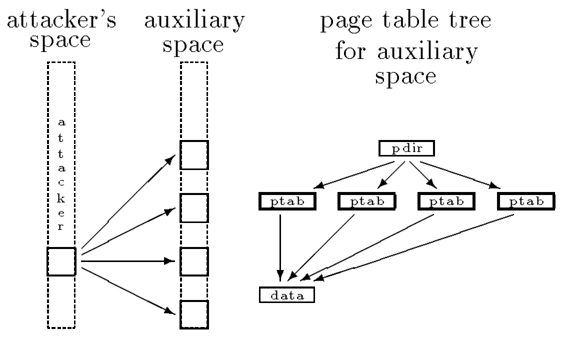

Figure 1: Mapping Attack

The basic problem is that μ-kernel and pagers compete for memory. When a pager issues a map or grant request, the new mapping can require additional page tables and/or an extension of the μ-kernel internal pmap tree table.

The μ-kernel maintains a pool of free page frames for its own use. Since the pages have been granted by the initial pager σ_0, no address space can contain them, not even σ_0. If the pool is exhausted and additional memory is required for one of the above mentioned operations, the μ-kernel denies the requested service: mapping, granting or thread creation simply fails with an appropriate error code.

Then the requestor must donate some of its memory to the μ-kernel or find some other pager donating memory to the μ-kernel.

The donation goes back to σ_0 which finally unmaps the page and grants it to the μ-kernel's page pool. Donating and granting the page is complemented by the information "dedicated to T", where T identifies the task that invoked the failing map/grant or create-thread operation. The μ-kernel attaches this information to the newly-received page and uses it thereafter only for mappings etc. requested by T.

Assume that the task T tries to map a page p into a client's address space. If the operation fails, T selects a page p', saves its contents if necessary, and gives it to the μ-kernel:

```
‘map’ in $T$:
    map ($p$, $dest$) ;
    while map failed do
        select ($p'$) ;
        unmap ($p'$) ; save if necessary ($p'$) ;
        rpc (next-level pager,
            “grant to $\mu$-kernel ($p'$ dedicated to $T$)”);
        map ($p$, $dest$)
    od .

‘grant to $\mu$-kernel’ in intermediate pager:
    unmap ($p'$) ; save if necessary ($p'$) ;
    rpc (next-level pager,
        “grant to $\mu$-kernel ($p'$ dedicated to $T$)”);
    reply (“done”) .

‘grant to $\mu$-kernel’ in $\sigma_0$:
    unmap ($p'$) ;
    grant ($p'$, $\mu$-kernel page pool, “dedicated to $T$”) ;
    reply (“done”) .
```

When the address space *dest* is deleted, all related page tables etc. are released and returned (granted) to σ_0. σ_0 should remember the next-level pager already donated the page to the μ-kernel. On request of this pager, σ_0 should remap it to this kernel etc. Finally, T could regain the page.

3.2 IO Ports and Interrupts

IO ports are managed like memory by pagers. The μ-kernel does not depend on their availability.

3.3 Threads and Tasks

Task creation and deletion is completely controllable by servers.

The number of threads per task is currently limited to 128. This number can be further reduced by the σ_1-pager which allocates pages for thread control blocks.

3.4 Processor and Time

A thread s with maximum controlled priority mcp_s can change priority and timeslice of a thread t provided for its current priority holds $p_t \leq mcp_s$. The new priority cannot exceed mcp_s. All threads of a task have the same *mcp*. When a new thread is created, the creator defines its new *mcp* which also cannot exceed the creator’s *mcp*.

Due to the priority system, threads with priorities $p_t > mcp_s$ are not affected. Related attacks can be handled outside the μ-kernel at the server level.

4 Server-Directed Attacks

Any of the mentioned μ-kernel resources could be attacked indirectly by attacking the server which manages the resource. Recall that attacking a server’s integrity is assumed to be blocked by the security system and will not be discussed here. Two types of attacks remain: either attack a secondary resource the server relies on (section 4.3) or attack the server by μ-kernel-provided inter-task operations.

By invoking inter-task operations, one task can influence another task. The μ-kernel provides four operations of this type (the remaining three system calls *thread_switch, nearest_id* and *lthread_ex_regs* affect only the invoker’s task and can thus not be used for attacks):

1. *task_new* permits to delete a task. However, only the creator of a task can delete it. Conclusion: The system call cannot be used for denial-of-service attacks.

2. *thread_schedule* permits to change priority and timeslice of thread’s in other tasks. However, the system call is only effective if the destination thread’s current priority is less or equal than the invoker’s maximum controlled priority, i.e. if the invoker has the explicit right to schedule the destination. Conclusion: The system call cannot be used for denial-of-service attacks.

3. *fpage_unmap* permits to unmap a page in a foreign address space. Assume that A tries to unmap a page in B. This requires that the page is also part of A’s address space and that it was already mapped (directly or indirectly) by A into B’s address space. In other words, B accepted a page mapping from A and therefore implicitly a subsequent unmap operation by A on the page. Conclusion: The system call cannot be used for denial-of-service attacks.

4. *ipc* permits to send messages across address spaces. Note that mapping and granting fpages is also done by ipc. Ipc requires a certain agreement between sender and receiver: the sender specifies the message to be sent; the receiver specifies (a) the receive buffer as well as the

type of message it is willing to receive and (b) whether it is willing to receive a message from this specific sender. However, there are only three receive states: 'not ready to receive', 'willing to receive from thread t' and 'willing to receive from any thread'. On the one hand, the latter status is required at least for general servers; on the other hand it might be used for attacks as well as for legal service requests.

Conclusion: The *ipc* system call is the only handle for denial-of-service attacks based on μ-kernel-provided inter-task operations. It permits pulsar-like attacks (section 4.1) and poisoning a receiver or sender with a malicious pager which never serves page faults (section 4.2).

4.1 Pulsar Attacks

The attacker sends junk messages as fast as possible to the attackee.

Messages are not buffered. Instead, the sending threads are put into a receiver-related queue. Therefore, the maximum queue length is basically the number n of attacking threads. Any legal request will be delayed at most by $n\epsilon$, where ϵ is the time required to transfer one message and to let the attackee classify it as a junk message. Defense possibilities:

a. The attack can only be made against server threads which frequently use open receive operations (receive from anyone). Therefore use one thread per client whenever possible. It can issue closed receive operations which makes this type of attack impossible.

b. Limit the receiver's buffer size for receive-from-any operations to limit ϵ.

c. Encapsulate suspicious clients by a clan. The chief can sequentialize the messages. Then, a pulsar attacks the chief instead of the servers outside the clan. For a more detailed description of this protection method see section 5.

4.2 Infinite Send/Receive

A sending thread may attack a receiving thread as shown in figure 2. The attacker sends a message from an unmapped region of its own address space. when the μ-kernel tries to copy the message into the receiver's address space, a page fault will be raised on the sender side. It has to be handled by the sender's

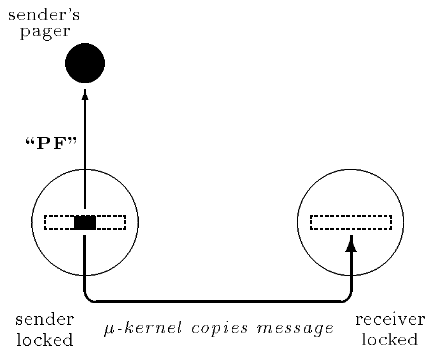

Figure 2: Sender Attacking Receiver

pager. If this one never resumes the fault the message transfer will never complete and the receiver will remain blocked forever. Defense:

a. The *receiver* specifies a *send-pagefault timeout.* The IPC operation is aborted if a page fault occurring in the sender's address space is not served within this receiver-specified timeout.

Symmetrically, a receiving thread can attack sending threads. The defense is then based on the *sender-specified* receive-pagefault timeout.

4.3 Secondary-Resource Denial

The servers managing μ-kernel related resources themselves need resources, in particular memory and processor time. Therefore the set of corresponding servers must be constructed in such a way that no trusted server depends on a less trusted server or a server not sufficiently robust against denial-of-service attacks. A way to ensure this:

a. Run all critical servers on a sufficiently high priority.

b. Give sufficient memory to all critical servers on initialization. Use a trusted pager which will never unmap this memory.

5 Using Clans for Defense

The L4 μ-kernel provides a general mechanism which can be used for defending against denial-of-service attacks: Clans & Chiefs [3]. A *clan* (denoted as an oval in figure 3) is a set of tasks (denoted as circles) headed by a *chief* task. Inside the clan all messages

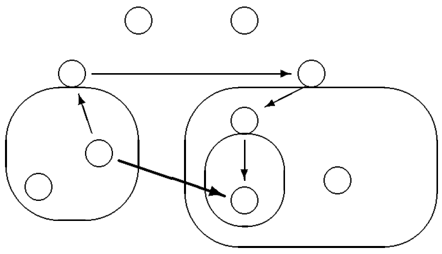

Figure 3: Clans & Chiefs

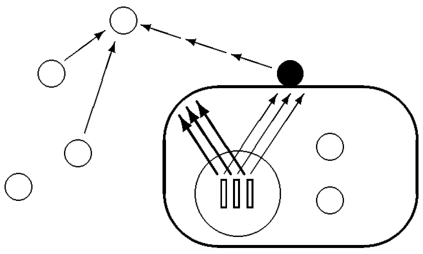

Figure 5: Anti-Pulsar Chief

are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. ("Original" ipc is denoted by thick lines, redirected ipc by thin lines.) The chief may inspect and/or modify the message. Clans can be nested.

Figure 4 shows a chief which is used to enforce the security policy. All server requests from the en-

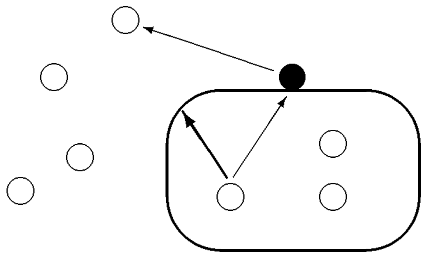

Figure 4: Security-Policy Chief

capsulated tasks are inspected by the chief (filled circle). The chief drops any request which would violate the security policy. In particular, it uses accounting mechanisms to restrict denial-of-service attacks. Note that page faults and mappings are also handled by IPC. Therefore the according resources are also under the chief's control.

Clans can also be used to restrict pulsar attacks 4.1. The chief sequentializes the messages of all threads running inside the clan's tasks. The vertical bars in figure 5 denote threads which together attempt to attack a server by flooding it with junk messages. Since the server is located outside the clan, the attacking threads are not queued at the server but at the anti-pulsar chief. The chief sends the IPC requests to the server, one after the other. As a result, never more than one instead of n junk IPCs compete with useful messages.

6 Conclusion

With the popularity of code downloading on the web operating must be structured to minimize and prevent the denial of service attacks. To this effect the operating system should be small in code size and should have a small set of interface calls.

We have shown that it is possible to construct and reason that denial of service attacks can be prevented in small μ-kernels. The burden falls on the servers to prevent attacks at a higher levels in the system's software.

References

[1] R. Anand, N. Islam, and J. R. Rao. A Capability–based Security Model for Using Internet Content. Technical Report 20664, IBM Research, 1996.

[2] M. Hohmuth, T. Jaegert, J. Liedtke, and J. Wolter. Guidelines for implementing os servers on top of L4. Research Report RC xxxxx, IBM T. J. Watson Research Center, to appear 1996.

[3] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[4] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[5] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996.

[6] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.