Analyzing Capsicum for Usability and Performance

Benjamin Farley

December 23, 2010

Abstract

In this paper I investigate Capsicum, an extension to UNIX that introduces a new security model on top of existing UNIX architecture. This model consists of several new security primitives and system calls that replace existing UNIX functionality. I focus on two aspects of Capsicum: performance and usability. For performance, I compare the performance of Capsicum system calls to corresponding UNIX calls and analyze these differences. I also implement a small file-hosting server that makes use of Capsicum's sandboxing library, in order to determine the feasibility of writing new applications using Capsicum and modifying existing applications to use Capsicum.

1 Introduction

Computer security is becoming an increasingly important problem in today's world. Viruses, worms, trojans and every other type of malware imaginable continue to propogate, despite the industry's best efforts to eradicate them. Furthermore, with recent large-scale attacks such as Stuxnet [2] and the 2008 denial of service attack on Georgia, cyber warfare is becoming a threat we must take into account when we design systems. Unfortunately, as virtually any user or developer of applications knows, security is also a very difficult problem to solve. Cyber attacks target software applications, or hardware, which are made and designed by programmers; thus, the security of an application is only as good as the developer can make it. For simple applications and programmers experienced in security, making a program secure is usually a reasonable task. However, the difficulty of this problem increases very quickly as software evolves and grows. As an application becomes bigger and more complex, it becomes increasingly difficult to understand it and determine how it will behave in a given situation. For especially large systems such as operating system kernels, which can reach millions of lines of code and are developed and supported by a large group of people, many of which are not well-versed in security, this problem becomes nearly impossible.

For these reasons, it is important that the security primitives provided by a system be able to take into account today's vastly complex software. In this work, I analyze a new system, Capsicum [6], with that goal in mind. In particular, I will focus on two aspects of the system: performance and usability. For performance, I measure the how long it takes various system calls to complete when used in capability mode versus in normal mode and analyze the differences I find. I also compare the performance of Capsicum calls to

corresponding UNIX calls. Finally, I create a basic server application and instrument it using Capsicum in order to understand how usable it is from an outside perspective and what changes need to be made to a normal application to let it make use of Capsicum functionality.

The remainder of this paper is organized as follows. Section 2 gives account of related work. Section 3 gives a brief overview of Capsicum's security model and the functionality it provides. In Section 4 I discuss the implementation of a real world application using Capsicum. Results are discussed in Section 5. Section 6 talks about some of the limitations I found in Capsicum and places where it could be improved. Finally, Section 7 gives my conclusions.

2 Related work

Object capability systems have been used in previous systems. For example, Google Caja [5] uses an object-capability model to attempt to secure JavaScript and online web browsing. It does so by creating and passing around objects which grant access to resources like web services. It also includes functionality to rewrite existing JavaScript programs to use capabilities. In [4], the authors analyze the security aspects of this capability-oriented security model. Wedge [1] uses a security model similar to the capabilities that Capsicum uses, but also includes the improved functionality of being able to control access to parts of memory on the heap.

Harris [3] discusses instrumenting existing programs for use in Flume, an OS based on the idea of decentralized information flow control (DIFC). In this system, security is achieved by controlling how information can flow between entities in a system. Harris discusses the problem of instrumenting complex, large programs with a small set of small security primitives in order to achieve a given high-level security policy.

3 Capsicum

Capsicum is a UNIX extension that provides a new security model on top of existing UNIX architecture. The basis for this model is the idea of a *capability*. A capability is an unforgeable token of authority which can be passed around between processes and is used to access resources. A capability can be thought of as an extension, or a wrapper, for basic UNIX file descriptors; it provides the same functionality they do and is used in much the same way, but also provides additional security and a fine-grained measure of control over the resource. For example, whereas traditionally one opens a file with simple read or write privileges, Capsicum provides over sixty options for specifying precisely how the resource will be used (including things like read, write, seek, fstat, etc).

One of the reasons Capsicum is relevant to this work is that capabilities provide an intuitive, powerful method of implementing security. In a capability system, holding a capability is sufficient to allow access to a resource. What this means is that once a process holds a capability for a given resource, the system does not need to do any additional security checks to make sure that the process can access that resource; the fact that the process has the capability means that it can access that resource. Conversely, if a process cannot access a resource it will not even know about the existence of said resource.

This is a useful property for several reasons. In terms of performance, it means that the

only time a process needs to take the time to do security checks is when a resource is first acquired. Thus, if any time-consuming checks do need to be done, they will usually be a one-time cost. Once a process has obtained the resources it needs, they should behave the same way as normal file descriptors. In terms of programmer usability, this means that, for most intents and purposes, a capability can simply be used as if it were a file descriptor. It will be acquired using Capsicum-specific syntax, but after that the developer and the application interact with it the same way they would with a normal file descriptor.

The other main primitive Capsicum provides is *capability mode*. Capability mode is denoted by a simple process credential flag, which processes can set through a new system call, cap_enter. Once in capability mode, a process has limited access to global namespaces such as the file system or the PID namespace. This means it cannot access files and has limited ability to communicate with other processes. However, a process inherits capabilities that it held before entering capability mode, as well as ones its parent process held.

Finally, the developers of Capsicum provided their own library, libcapsicum, that uses the above to provide higher level functionality, with a focus on sandboxing. This library was what I used to instrument my example program, and as such will be the focus of my evaluation.

4 Implementation

To evaluate Capsicum on the above criteria, I wrote a basic file-hosting server. This server lets clients store and access files in a secure manner. Specifically, the security policy I chose to implement was that clients could access only their own files. They could not access the files of other clients, and could not see any of the filesystem except their own directory. In this model, I make the assumption that workers are untrusted processes. Thus, we can assume that a worker thread may be compromised but if it is, the security policy of the system still must not be violated. The implementation of this server in a normal setting is straightforward, and works pretty much exactly how you would expect it to. The flow of events is summarized in Figure 1. The server accepts incoming connections from clients and spawns a worker thread to deal with each request it receives. After the initial request, most interaction occurs between the client and the worker. The only aspect of my implementation that may be slightly surprising is that the server spawns threads upon receiving requests, rather than using a thread pool. Since thread pools are often a better solution for client/server needs, it seems that it would make more sense to use this in my implementation. Unfortunately, it turns out that, due to limitations in Capsicum, a thread pool does not work well for this particular scenario. This will be discussed further in Section 6.

From here, I used libcapsicum to instrument this server to use Capsicum's security functionality. In doing this, I hoped to understand what changes need to be made to a normal program to get it to work with Capsicum, and to analyze how usable Capsicum is for developers from an outside perspective. I also wanted to determine what kind of performance difference a real world application would experience when using Capsicum. Figure 2 shows the flow of the modified server. As the diagrams show, the instrumented application runs much the same way as the original. The main difference is that with Cap-

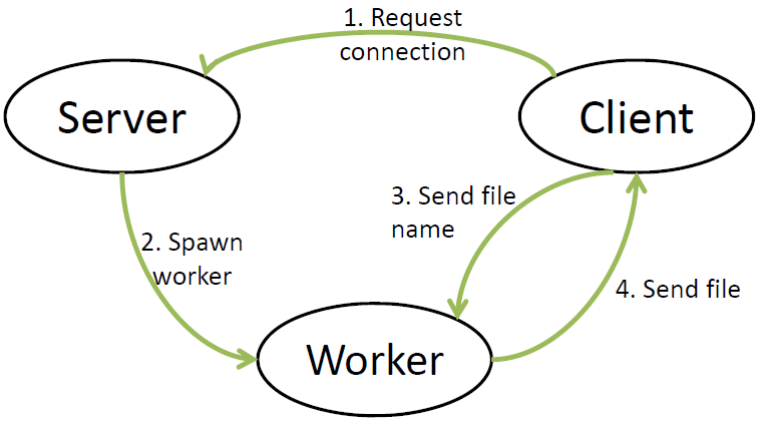

Figure 1: The flow of events in the normal server

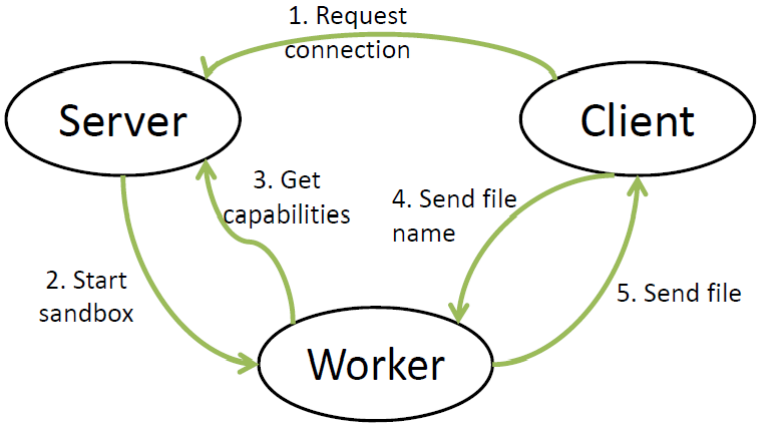

Figure 2: The flow of events in the Capsicum server

sicum, the worker must do an extra call to obtain the correct capabilities from the server before it can begin to talk to the client. My findings while making these changes will be discussed in Section 5.

5 Results

We can now turn to the results obtained from the above implementation.

Performance Figure 3 shows the performance of system calls in Capsicum compared to normal UNIX. There are some stark differences here, specifically in the calls to open and create files. These calls take around two to three times longer in Capsicum than in normal mode. This is a significant difference, but an understandable one. As mentioned previously, with capabilities most security checks must be done upon acquiring a resource. The system must verify that the process is able to access the given resource it is requesting. However, once the resource is acquired, the process can treat the capability the same way it would a normal file descriptor, and can expect it to act the same way.

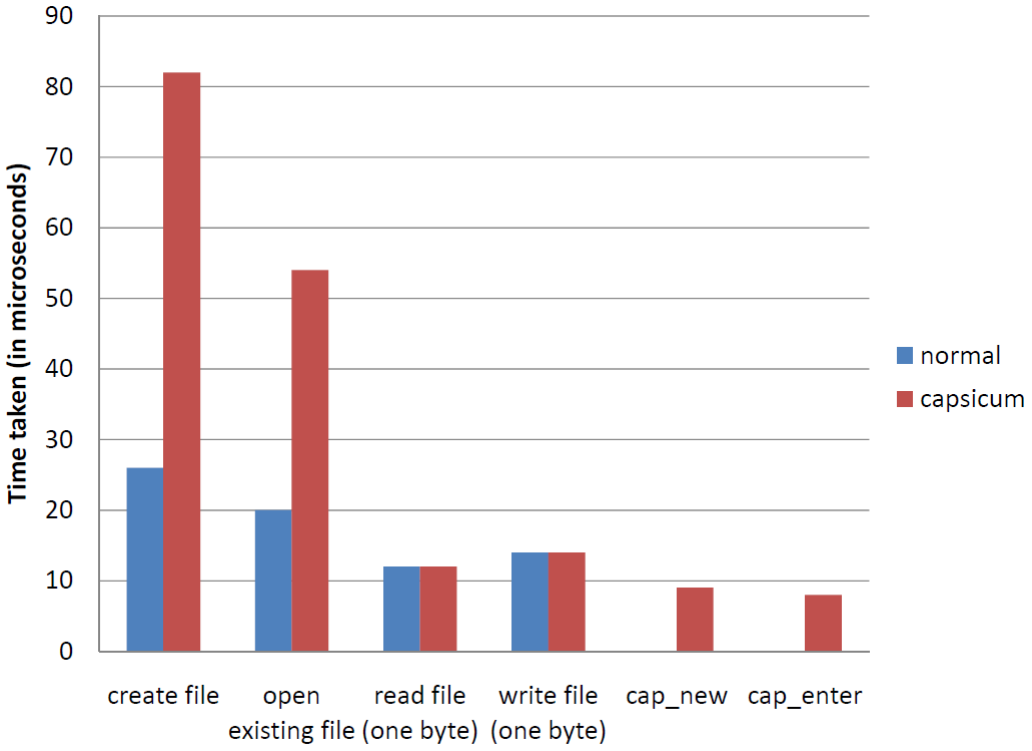

Figure 3: Performance of various system calls in normal mode compared to the same calls in capability mode.

We can see this in the bars for reading and writing a file. Here, there is virtually no difference between Capsicum and normal calls. Again, now that the process has acquired the resource, there are no extraneous security checks. The process possesses the capability; therefore, it can access the resource. This is one of the nice properties of capabilities - once they are acquired, they enforce security with minimal additional overhead. The final two bars in Figure 3 show Capsicum's performance on two basic calls, cap_new and cap_enter, which create a new capability out of a file descriptor and enter capability mode, respectively. These are both turn out to be quick calls at around eight to nine microseconds each, which is a reasonable amount of overhead for such simple calls. The call to enter capability mode is actually quite reasonable, seeing as it can only be called once per process; a ten microsecond overhead once per program is virtually unnoticeable.

In Figure 4 we can see the performance of both systems when spawning new processes. For UNIX, I measured the time taken for a simple fork call. For Capsicum, I measured the performance of the lch_start() call. This is part of libcapsicum, and is simply a way of setting up a sandboxed environment for a new process to run in. Here we once again see a fairly significant performance hit, around 50% increase for Capsicum. Again, this is fairly intuitive. Capsicum is setting up an entire sandboxed environment, limiting its access to only certain resources, and spawning a new process in that sandbox, so it makes sense that it takes longer to do this than to simply fork a new process.

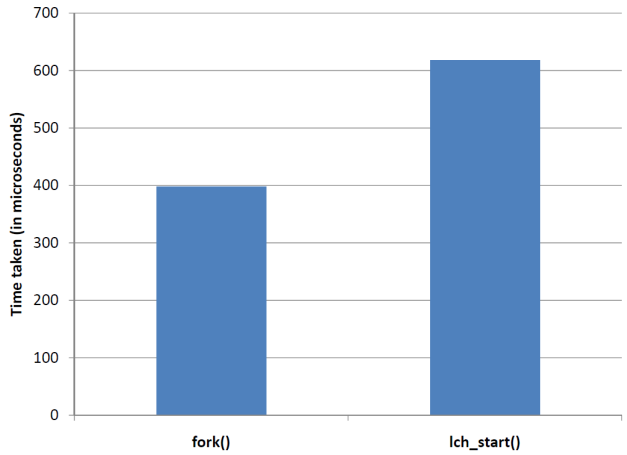

Figure 4: Spawning a new process. This compares the UNIX fork() call to Capsicum's lch_start(), used to spawn a new process in a sandbox.

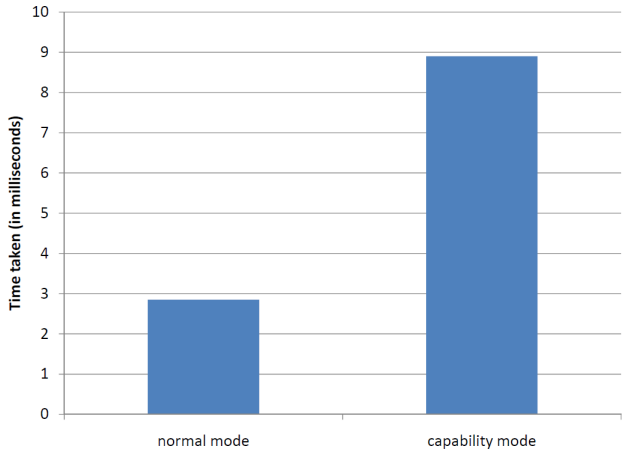

Figure 5: Retrieving a single file from the Capsicum server and the normal server.

Finally, let us turn to the performance of the real world server application. Figure 5 shows the time taken for a client to retrieve a single file from the Capsicum server versus how long it takes to retrieve the same file from the normal server. Here we see a huge difference, around three milliseconds for the normal server compared to almost nine milliseconds for the Capsicum server. Based on the above measurements, this seems excessive. We saw some overhead for some calls, but nothing that would cause nearly six milliseconds of overhead. To figure out this anomaly, it is useful to break the flow of the server down into its basic parts and analyze them individually. The flow for the server and worker can be found in Table 1. As we can see, even with the extra calls Capsicum makes, the time it takes a Capsicum worker to serve the file is barely more than a millisecond longer than it takes the normal server.

Clearly, it is not the worker itself that is causing this performance hit. The issue becomes more clear when we look at the flow of

	Normal server	**Capsicum server**
server: open directory	45	33
server: create fdlist	n/a	85
server: spawn worker	398	618
worker: get capabilities	n/a	862
worker: ACK client	182	171
worker: read filename	324	305
worker: open file	85	73
worker: read from file	21	18
worker: send file	158	159
total	**1213**	**2324**

Table 1: Server and worker flow. This table shows the time taken (in microseconds) for all actions taken by the server and worker at each step of the process in retrieving a file.

	Normal server	**Capsicum server**
send request	132	129
wait ACK from worker	1429	7854
send filename	134	134
receive file	722	688

Table 2: Client flow. This table shows the time taken (in microseconds) at each step for the client in receiving a file.

the client program, shown in Table 2. Here we see that, of those nine milliseconds, almost eight of them are spent waiting for the worker to begin communication with the client. This implies that the measurement we saw in Figure 4 for time taken to start a sandbox is misleading. It turns out that that measurement is simply the time it takens for control to return to the host process, and it takes much longer for the sandboxed process to actually begin running.

Though these results may seem disappointing, they are actually not all bad news. Most of the calls that suffer significant time delays are calls that are only done a few times in a program, such as creating and opening files. The problem with overhead during sandbox creation is more serious, but this is also a call that will usually only be done a few times per program. Many calls that will be used more often, like read and write, suffer no overhead when used in capability mode.

Usability A large part of my goal in implementing a server using Capsicum was to see how usable Capsicum is in the eyes of a programmer who has never used the system before. Given a program written for a normal system, how much do I have to change it to make it work with Capsicum? I came to several conclusions in this regard. Firstly, there is a significant amount of code that does not need to be touched. The entire client portion of the application remained unchanged; the

same client could interact easily with either server. Furthermore, for the most part, the worker was also unchanged. As we saw in Figure 2, the main difference between the normal worker and the Capsicum worker was that the Capsicum worker had to retrieve capabilities from the server before communicating with the client. This required a small change at the beginning of the worker, around five lines to get the capabilities and initialize the correct pointers. After that, the worker code for the two systems was identical. Similarly, the only difference between the servers was the way in which they spawned a new process; fork() for the normal server and lch_start() for the Capsicum server. To use this call, the Capsicum version also needed to add around ten lines of code to initialize sandbox varialbles; other than that, the servers were identical.

These are promising results. The fact that capabilities are so similar to file descriptors in most uses means that the majority of existing code does not have to change. Calls that access a file descriptor can access a capability in the exact same way. Communication via sockets is unchanged. In nearly all cases, the only changes that need to be made to a program are in the places where it acquires resources. This means that a developer who wants to add Capsicum functionality to a program does not have to rework his entire code base, and that a developer who is writing a new program can worry about the security aspects of the program at specific points and then forget about them.

6 Limitations

Capsicum is research software, still under development, and as such is not nearly perfect. In performing these experiments, I found several ways in which limitations in the Capsicum implementation prevented me from achieving a desired goal. The most important of these is that Capsicum does not provide a way to revoke capabilities from a process; once a process has a capability to access a given resource, it will always have that capability. We can illustrate why this is a problem by going back to the problem mentioned earlier having to do with thread pools. With thread pools, a server would follow a similar flow to that shown in Figures 1 and 2. However, after completing the request, instead of exiting, the worker would retrieve another job from the server. This poses a problem for Capsicum, given the security policy that clients must not be able to access each others' files. After serving a request from a user, a worker thread has the capabilities corresponding to that user's files. If the worker immediately goes to retrieve another job, it will then begin interaction with the next user while still holding the original capabilities. This means that the next user could access the first user's files. Unfortunately, Capsicum does not provide a means of preventing this, as capabilities cannot be revoked.

There are other minor limitations that I found in Capsicum. Some of the implementation is somewhat ad-hoc or even lazy. Many of the calls Capsicum provides, such as sending capabilities between host processes and the sandboxes they have started, are simply piggy-packed on top of existing UNIX calls. Thus there are often unnecessary or redundant parameters or calls made in performing these actions. While this is an easy solution to the problem of passing capabilities, it could perhaps be solved more cleanly and efficiently if a dedicated capability passing scheme was used instead.

7 Conclusions

In this work I analyzed Capsicum to determine how it performs compared to existing UNIX architecture and how usable it is from a developers point of view. I found that in terms of performance, there are both good and bad aspects of Capsicum. Many calls function exactly as they do in a normal system, with little to no overhead for common calls like read and write. Unfortunately, some calls entail significant overhead, much of which us created by Capsicum's under-the-hood manipulations and is not immediately visible to the programmer. Fortunately, the calls that introduce overhead are usually calls that only need to be made once, such as entering a sandbox or creating a file.

In terms of usability, Capsicum again has pros and cons. Though it has a complicated call structure and occasionally seems to make arbitrary design decisions, it is fairly straightforward to use once a programmer learns its syntax and calling conventions. Furthermore, the majority of Capsicum's primitives deal with a small percentage of the total code in a program, which means that to modify a program to use Capsicum, a programmer can get away with looking at only a limited portion of the code. With all this taken into account, Capsicum is a promising system that has potential to be quite useful in the security community.

References

[1] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. *5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[2] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2010.

[3] William R. Harris, Thomas Reps, and Somesh Jha. Difc programs by automatic instrumentation. *Computer and Communications Security*, 2010.

[4] Leo A. Meyerovich, A. Porter Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. *World Wide Web Conference*, 2010.

[5] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized javascript. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf, October 2007.

[6] Robert Watson. Capsicum: practical capabilities for unix. *19th USENIX Security Symposium*, 2010.