Toward an Automated Vulnerability Comparison of Open Source IMAP Servers

Chaos Golubitsky – Carnegie Mellon University

ABSTRACT

The attack surface concept provides a means of discussing the susceptibility of software to as-yet-unknown attacks. A system's attack surface encompasses the methods the system makes available to an attacker, and the system resources which can be used to further an attack. A measurement of the size of the attack surface could be used to compare the security of multiple systems which perform the same function.

The Internet Message Access Protocol (IMAP) has been in existence for over a decade. Relative to HTTP or SMTP, IMAP is a niche protocol, but IMAP servers are widely deployed nonetheless. There are three popular open source UNIX IMAP servers – UW-IMAP, Cyrus, and Courier-IMAP – and there has not been a formal security comparison between them.

In this paper, I use attack surfaces to compare the relative security risks posed by these three products. I undertake this evaluation in service of two complementary goals: to provide an honest examination of the security postures and risks of the three servers, and to advance the study of attack surfaces by performing an automated attack surface measurement using a methodology based on counting entry and exit points in the code.

Introduction

System administrators frequently confront the problem of selecting a software package to perform a desired function. Many considerations affect this decision, including functionality, ease of installation, software support, interoperability, performance, and hardware and software dependencies. However, the sensible administrator will place significant weight on choosing a software package with a reasonable posture towards security.

When selecting a package and installing it initially, the administrator can ensure relatively easily that the new configuration is not susceptible to any known vulnerabilities. However, some software is more prone to future attacks than others. Sysadmins could benefit from an easy-to-implement metric which provides an intelligent assessment of which software packages are likely to be found vulnerable in the future, not just of which ones have had problems in the past.

Measurement of the attack surface of a software package has been proposed as such a metric. The attack surface is the set of opportunities which the software package provides for the outside world to access the software itself. It is measured by enumerating this set and classifying its elements on the basis of risk of future attack. Researchers have further proposed automating attack surface measurement by enumerating and classifying the entry and exit points of the software package. In this paper, I describe steps taken towards performing an automated relative vulnerability assessment of open source UNIX IMAP servers based on measurement of attack surfaces.

Contributions and Roadmap

The paper makes two major contributions. First, I undertake an in-depth discussion of the relative security postures of the three major open source IMAP servers in use today. Second, I perform a partial measurement of attack surfaces based on the entry/exit point methodology, and compare the results to my initial impressions of the servers studied.

In the remainder of this section, I introduce attack surfaces and IMAP servers. I discuss the prior work done on attack surface measurement and the rationale for selecting IMAP servers as the target of my analysis. I discuss IMAP servers at the design level by first exploring the high-level interactions of the servers with their environments, then introducing the three target servers with a focus on the design philosophies and implementation peculiarities of each. In the next section, I introduce the entry/exit point method of attack surface measurement, and discuss the methodology used in performing a partial entry/exit analysis of the IMAP servers, including roadblocks to performance of a fully automated analysis. Finally, I compare the results of the analyses performed, and provide conclusions and future directions for attack surface research.

Attack Surfaces

Theory of Attack Surfaces

The attack surface can be used to create a comparative software vulnerability metric for packages which perform similar functions. Such a metric assesses a system at the design level by observing the prerequisites for a system to be attacked.

An attack on a system necessarily involves an attacker who gains access to some resource to which

he is not entitled. There are some prerequisites without which an attack cannot occur. First, the attacker must be able to take some action which affects the system. If he cannot alter a system's state at all, he cannot increase his access to that system. Second, in order for an attack to be non-trivial, the system must contain some resource which is accessible after the attack, but not before. This resource may itself be the target of the attack, or it may be an enabler which allows the attacker to reach his eventual target, or to get closer to it. For example, in the case in which an unauthenticated attacker triggers a buffer overflow in a port-listening daemon to obtain a root shell on the server, the action is the daemon, which runs code based on connections received from anyone, and the resource is the daemon's root privilege.

The attack surface of a system is simply the set of all actions made available by the system and all resources accessible to the system. This set consists of three types of items: (1) methods, which are executable code potentially runnable by an attacker, (2) channels, which are IPC mechanisms potentially usable by an attacker, and (3) data items, which are sources of persistent data (such as files) potentially readable or writable by an attacker.

However, enumerating system actions and resources is not very useful in itself, because it tells us nothing about the specific risks posed by the actions and the specific opportunities afforded by the resources. A port-listening daemon which requires connections to come from a specific IP address and which asks clients to cryptographically authenticate themselves before communicating further may pose less risk than does a daemon which will allow any client to talk to it. We need some way to discuss the relative risks posed by different methods or resources.

Prior Work on Attack Surface Measurement

The idea of attack surfaces was introduced by Howard, and was used to measure the relative attack surfaces of versions of Windows [11]. Manadhata and Wing extended the description of attack surfaces in 2004, and discussed ways of measuring attack surfaces in the context of Linux versions [14]. Their approach to the problem that not all system resources are created equal was to use the Common Vulnerabilities and Exposures (CVE) database [1] to identify several types of objects which were most often involved in attacks on Linux. Typed objects include things like "http daemon running as root" or "file writable by group users." They then enumerated those types within instances of each of several Linux distributions to draw a comparison.

Several improvements on their analysis are envisioned here. First, manual analysis is tedious and error-prone, so a means of automating attack surface measurement is desired. Second, it is not possible to use this method to compare attack surfaces between two products – if one has more daemons running as root, but the other has more world-writable files, we cannot say anything further about which poses the greater risk. Third, identifying important resources based on previously discovered vulnerabilities assumes both that the set of known vulnerabilities is large enough to be representative[1] and that future attacks will exploit the same types of resources and actions as previous attacks. In order to focus on features which could be attacked in the future, we need to define types in a way which more accurately assesses the risk introduced by each. The entry/exit point analysis discussed in the third section of this paper attempts to address these issues.

IMAP Servers

The Internet Message Access Protocol provides e-mail access to authenticated remote users. Once a Mail Transfer Agent receives a message for a local recipient, it must place that message into a data store from which the recipient can access it at his leisure. In many cases, the MTA places the message directly into a file, using a mail storage format such as Maildir or Berkeley mbox. The recipient then logs onto a UNIX system which has access to the mail file, and uses a UNIX-based mail client (MUA) to read the message.

IMAP provides a mechanism for the mail store to be accessed over a network. The MTA hands a message off to the IMAP server, either by storing it in a file format the server can read, or by communicating with the IMAP server using a mail transmission protocol such as SMTP or LMTP (Local Mail Transfer Protocol). At a later time, the recipient may connect to the IMAP server over a network, authenticate, and perform actions such as reading new messages, deleting messages, or selecting messages from his mailbox based on certain criteria [15].

I chose to investigate IMAP for several reasons. First, there are three freely-available open source IMAP servers which share the bulk of the market – UW-IMAP, Cyrus, and Courier-IMAP. Each server has been in development for several years, so the codebases are fairly mature. Each codebase contains 100,000-250,000 lines of code. These codebases are large enough that it is not practical to perform an analysis by reading every line of the code, but small enough that it is possible to get a design-level sense of the actions the code takes. Second, since IMAP is a niche protocol relative to, for instance, SMTP or HTTP, there has not been a formal security analysis of IMAP servers in the past. Third, the IMAP protocol is stateful, and, in particular, provides for both authenticated and unauthenticated states. In theory, attack surfaces are well suited for analysing stateful protocols, because they take into account what privileges are prerequisites for attacking a certain portion of a system. For all these reasons, studying the attack surfaces of IMAP

[1] This is certainly not the case for IMAP servers, for which only about 30 distinct server vulnerabilities are reported in the CVE.

servers is both feasible and worthwhile, in that it provides useful information about server security.

Observation of IMAP Server Software

To assess the effectiveness of the attack surface analysis, I needed to gain a subjective impression of the relative security postures of the three IMAP server packages. In order both to increase my knowledge of IMAP server operation, and to build a consistent baseline from which to compare the codebases, I installed each server package in a constrained environment. To the extent possible, I applied the same minimal configuration to each server.

Observation Methodology

I installed each package using the jail() system call under FreeBSD 5.2. The jail() call is an extension of chroot() which restricts a called program and its children to operating within a subdirectory of the filesystem. All dependencies of the jailed program, including libraries, configuration files, devices, and log files and sockets, must exist within that subdirectory. In addition, jail() binds all network activity of the jailed process and its children to one given IP address [12].

Using jail(), I was able to create a miniature virtual server for each IMAP package. Each virtual server ran only three items: a syslog daemon providing debug-level system logging for all activity, a listener of some sort to receive e-mail via LMTP connections, and the IMAP daemon itself. In addition, I created a virtual server containing the Postfix MTA, whose purpose was to forward mail via LMTP to each of the three IMAP servers.

For each IMAP server, I attempted to create a minimal default configuration in which the server listened only on port 143 for both encrypted and unencrypted connections.[2] Each server provided access to one user account, which could be authenticated via a CRAM-MD5 database.

I determined the minimal set of files which each server needed in order to function by trial and error. I modified the jail's startup routine to start all programs as children of ktrace(), and used that in combination with syslog error output to iteratively determine what additional files were needed. Needless to say, this procedure does not give an entirely comprehensive view of all the files a given program might use under any circumstances. However, it does enable some subjective assessment of the behavior of each package, including: installed size and dependencies, behavior during compilation and configuration, network and system behavior during operation, and any notable extra features the software provided, be these user-visible options, extra daemons and port listeners, or API methods.

[2] Technically, the port listens only for unencrypted connections. However, it is possible for a client to connect and immediately issue the STARTTLS command, thereby encrypting the remainder of the session.

In the remainder of this section, I will introduce each of the three IMAP servers in turn, discussing the project overview of the package, the code layout and high-level design, my experience in installing and configuring the package, and my subjective impression of the security of the package.

High-Level Interactions of IMAP Servers

Some high-level elements are common to all IMAP servers, as a result of the services they provide. In particular, there are several ways in which an IMAP server must interact with its environment in order to do its job. These interactions correspond to types of attack surface elements which all IMAP servers must have, so it is worthwhile to outline them in a general sense before proceeding to the specifics of each server package. Figure 1 depicts the interactions described in the remainder of this section.

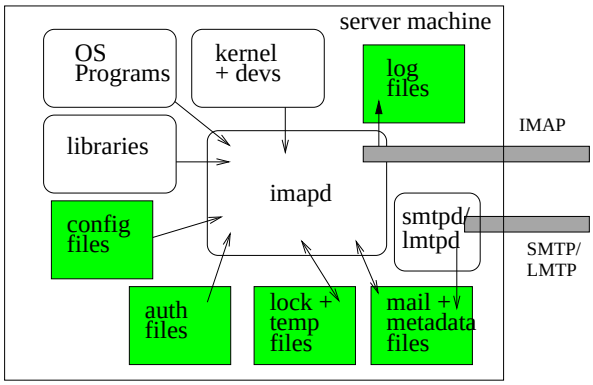

Figure 1: High-level IMAP server interactions and dependencies. (Executables are white, files are grey. Imapd may provide lmtpd service internally).

Remote API

First and foremost, the IMAP server provides a network API (application program interface) to a remote user. The server listens on a port – commonly port 143 for unencrypted or encrypted connections, or port 993 for encrypted connections only – and responds to user connection requests.

The IMAP API is the set of commands which the IMAP server considers to be legal for the remote client to send. It consists of 30-50 commands, the majority of which are specified by the IMAP RFC. The user sends these commands to the server using a fixed format and a command-specific number of arguments, and each command has an RFC-defined effect. The command set includes actions such as CAPABILITY (list the features offered by this IMAP server), STARTTLS (negotiate encryption for this connection), AUTHENTICATE (begin some form of negotiation to move from an unauthenticated state to an authenticated state), and SELECT (choose a mailbox for further operations).

Most of the commands deal directly with processing of e-mail, and thus should be accessible only

to authenticated users. However, a set of authentication commands must be available to anyone who connects to the server, so that authentication can take place. In addition, some servers offer support for folders available to anonymous users, and thus for an anonymous login mechanism.[3]

Access to User E-Mail Data

In order to serve e-mail, the IMAP server must have a way of receiving that mail from the MTA. The e-mail reception mechanism can take one of two forms: either the MTA deposits the mail in a format the IMAP server can read, or the IMAP server listens over a network or named socket for connections from the MTA. In the former case, the MTA and the IMAP server must share disk access. In the latter case, the IMAP server must run another listening daemon which speaks a mail transfer protocol such as SMTP or LMTP.

Once the mail has been received, the IMAP server must be able to read and modify the messages on disk, and must maintain e-mail metadata in order to keep track of mailbox state required by the IMAP protocol. Typically, metadata is stored as separate files associated with each user or each mailbox, or as header data within mail files.

Operating System Dependencies

The IMAP server depends on its resident operating system (a version of UNIX in the cases of my three reference servers) to provide it with several necessary functions. First, there is a standard set of utilities and system libraries – such as ls and libc – which are needed by almost any running program.

Second, the IMAP server needs a way to listen for incoming connections over the network. Some servers implement their own networking code, while some take advantage of OS-provided daemons such as inetd to launch the IMAP server in response to connection requests.

Third, each IMAP server uses some third-party libraries to provide various functions. All three servers make use of OpenSSL to encrypt the communication channel with the remote user. In addition, some of the servers use libraries such as BerkeleyDB to process authentication or metadata files in database formats.

Access to Non-Mail Files

Lastly, each server makes use of a number of files which are not directly accessible to the remote user or related to e-mail processing, but are used by the server backend. These files are of four primary types: (1) configuration files to be read at server startup or during operation, (2) log files or a logging subsystem to be written with error and/or debugging output, (3) authentication files or an authentication subsystem to be consulted when a remote user attempts to prove his identity, and (4) lock files and temporary files to be written in order to store non-permanent data or to control access to shared elements.

[3]This mechanism typically allows for login with the username anonymous and any valid text string as a password, and should be familiar to users of FTP.

UW-IMAP Server

Overview

UW-IMAP [9] is written and maintained at the University of Washington by Mark Crispin, the author of the original IMAP RFC. The purpose of this package is to provide a simple and flexible drop-in IMAP server for multi-user systems. The package uses the assumption that IMAP will be one of many login methods through which remote users can access the system. In particular, the functional differences between IMAP access and a shell access method such as SSH should be only that IMAP access is optimized for mail reading. Restricting IMAP access beyond the access afforded to a shell user is not a design goal.

The UW-IMAP server has been under active development since 1988, though the entire codebase has been rewritten several times since then. The current code is considered to go back only as far as the 2000 imap-2000 release. Looking further back, I find a code overlap of approximately 20% between imap-2004c1 (the most recent version as of this writing) and the 1996 imap-4 release, and no overlap between imap-2004c1 and any release prior to imap-4.

The current codebase contains 135,000 lines of code and 40,000 lines of other files. Of this code, the IMAP server itself comprises only 4,000 lines, while the remainder of the code consists of an internal (compiled-in) library called c-client. This library is also the backend for the Pine e-mail client.

Compiling imapd provides a single binary with a single purpose. An external program such as inetd must be used to listen on the appropriate IMAP ports. When a connection is made, an imapd process is spawned, handles that single connection, then terminates. Since UW imapd's place in the system is simple, the amount of code needed for its implementation is reduced. The tradeoff is increased dependencies on other programs to perform core functions, most notably mail delivery and port listening. The imapd program also requires no configuration file – configuration options are to be selected at compile time.

One more notable feature of UW-IMAP is that it is agnostic about mailbox formats. By default, the UNIX UW installation is compiled with support for mbox, mbx, mx, mh, tenex, mtx, mmdf, and phile mailbox types. This support is provided by means of mailbox drivers. Internal logic is used to guess the type of a mailbox, and then execution is passed off to the appropriate driver.

Impressions

The UW codebase does not have a good security history, and the design does not inspire confidence about

its future. My survey of IMAP vulnerabilities found in the CVE database (through December 2004) found twelve relevant vulnerabilities in the UW server, compared to seven in Cyrus and four in Courier. UW-IMAP's primary benefit is that it is the smallest and simplest of the three servers, both in terms of code size and major functions provided, and in that it provides a smaller set of IMAP API methods than the other servers. (The small API set may be in part due to the fact that the UW author wrote the IMAP RFC, which defines the minimal allowable set of API functions.)

However, the drawbacks are many, and seem to go down to the design philosophy of the package. The code is not at all modular. An example of this is given by the imapd program's main() routine. The routine contains code to select a mailbox sorting function for the SORT API method. It also contains three separate code locations at which the anonymous user is configured (a pre-authentication check at the beginning of the connection, an outcome of the LOGIN API method, and an outcome of the AUTHENTICATE API method), causing three distinct variables to be set each time. In addition, since the codebase is so non-modular, and since most of the functionality is provided by a c-client library which is also the backend for the mail client Pine, it is possible that functionality may be compiled in to the UW server which is really only necessary or desirable for client operation.

The design decision that IMAP access should be only one method into a user-accessible system effectively prevents administrators from building a closed box IMAP server. (UW can be compiled in a mode called closedBox, but the differences from the standard configuration are limited, and this is not an officially supported configuration.) The assumption that the server will not be a closed box shows up in several places. For example, one of the methods by which the Pine client attempts to contact an IMAP server uses rsh or ssh to run a server instance locally. In order to support this, the imapd server offers a ''pre-authenticated'' mode, in which a server started under a given UID is assumed to be authenticated as the associated IMAP user.

Despite UW-IMAP's history of buffer overflows, instances of string functions which do not perform length-checking (such as sprintf) are still plentiful within the code. The codebase uses custom makefiles, rather than GNU configure or some other standard solution, to maintain the cross-platform build process. This choice is not necessarily related to security, but it causes code compilation to be less uniform than might otherwise be the case. Along the lines of the closedBox option, the package officially disallows use of a configuration file, but the code specifies the name of a configuration file which will be read if it exists. According to the documentation, the results of using this file to configure the software are unpredictable.

Cyrus Server

Overview

Cyrus is written and maintained by Project Cyrus at Carnegie Mellon University [4]. Its purpose is to provide fast and scalable IMAP service. In order to support this goal effectively, the Cyrus server is, by design, truly a closed box. Cyrus permissions are defined internally and do not map to the UNIX permissions of the host operating system, and Cyrus uses its own custom mailbox format which is not guaranteed to be parsable by non-Cyrus tools.

Cyrus has been under active development since 1994. In approximately 1999, the codebase was split into cyrus-imap, the IMAP server, and cyrus-sasl, a set of flexible authentication libraries and associated utilities for use with IMAP or other Cyrus programs.

The Cyrus codebase contains 210,000 lines of code and 475,000 lines of other files. It is therefore the bulkiest of the three codebases, but is also relatively well documented. Of the included code, 75,000 lines come from the SASL codebase (the wrapper authentication libraries themselves, the optional authentication daemon, plugins for common authentication mechanisms, and utilities for checking and changing passwords), while the remainder is the IMAP codebase. The IMAP side of the code provides a number of auxiliary tools and functions, but the code specific to imapd itself is nearly 70,000 lines.

Compiling IMAP and SASL produces a large number of binaries and libraries, many of them optional. In order to function at all, the system requires: the front-end daemon master to listen for incoming network connections and pass them off to the appropriate servers, the imapd binary to handle IMAP connections, the lmtpd binary to receive incoming mail from the MTA and store it in the Cyrus mail format, and the ctl_cyrusdb binary to maintain Cyrus's extensive collection of mail metadata. The system also requires the primary SASL library libsasl2, as well as some helper libraries which implement particular authentication mechanisms. Cyrus is configured using the files cyrus.conf, which specifies the listeners and periodic processes spawned by master, and imapd.conf, which configures IMAP-specific options.

In order to support fast and scalable service, all Cyrus code runs under a single UNIX user account. It is possible to use the UNIX passwd file for authentication of IMAP users, but doing so requires the use of a special daemon, saslauthd, which runs as root and with which the SASL authentication mechanisms communicate via a named socket.

The other major Cyrus design decision which contributes to scalability is the custom mailbox format. Individual e-mail messages are stored in a readable format in flat files, one file per message contained in one directory per mail folder. (This directory

structure is similar to that of Maildir.) Mail metadata is stored in binary database files using a Cyrus-specific format, and thus cannot necessarily be read by non-Cyrus tools.

These two design choices bring with them the requirement that all access to or support of user mail be performed using Cyrus tools, and typically over the network. As a result, mail administration commands are provided via the remote IMAP interface, and a special subsystem, called sieve,[4] is provided for end user mail filtering tasks which other systems handle by allowing users to install procmail filters. When installed, sieve adds extra code, as well as some mechanism for end users to edit their filters – the most common is an extra port listener through which users can authenticate and make changes over the network.

Impressions

I found Cyrus to be the most impressive of the codebases in terms of layout. The code is modular and well documented, and can be easily compiled using GNU configure.

Since the IMAP server is designed to be run in a black box, any functionality not explicitly provided should be denied. Therefore, Project Cyrus takes breaches very seriously. From a security perspective, the design feature by which Cyrus runs all its code as the imapd user has both a major benefit and a serious drawback. The benefit is that no code (other than the optional saslauthd) runs as root, so it should never be possible to gain root access on the Cyrus box solely by compromising the Cyrus server. Therefore, if your primary concern is the prevention of root compromises on your network, the Cyrus methodology is a very good one. However, the downside is that any arbitrary code execution vulnerability constitutes a total breach of the IMAP system. If your primary concern is the safety of the mail system itself, you might prefer a system in which a buffer overflow in post-authentication code gave the authenticated user access only to his own mail, rather than to all mail and mail metadata on the system.

Cyrus provides a large number of features internally, some of which (such as mail administration and filtering) are required by the design of the system, but some of which seem like evidence of feature creep. For instance, there is a notification daemon to provide flexible user notification of new e-mail. Additionally, Cyrus, like UW, has a built-in NNTP client, and provides anonymous user access for the purpose of using the IMAP server as an NNTP aggregator.

Obviously, the in-band mail administration is itself risky, since it opens up more code to the IMAP interface. This risk is somewhat mitigated by the clean and modular design of the IMAP-visible code, but cannot be entirely removed. The Cyrus system is also busier than the others – the master process spawns periodic tasks related to cleaning and optimizing the database, while UW and Courier's daemons perform no work unless a connection is being served.

[4] Sieve is an open standard mail filtering language, defined by RFC 3028.

Courier-IMAP Server

Overview

Courier-IMAP is the IMAP server component of the Courier MTA [2]. The IMAP server is packaged with the MTA, but is also available and configurable as a stand-alone server. Courier is designed to be fast and scalable while being interoperable with standard UNIX permission schemes and mailbox formats. It attains this balance by using the Maildir format, which is a one-message-per-file format similar to Cyrus, but is supported by many other applications, including several MTAs.

Courier-IMAP has been in development since 1998, and has been a continuous codebase since that time. Like Cyrus, Courier is divided into an IMAP portion of the codebase and an authentication library, called courier-authlib. However, the imap/authlib split was recent in this case, dating back only as far as the 2004 release of courier-imap-4.0.0.

The current codebase contains 135,000 lines of code and 550,000 lines of other files, so it is relatively lean and also well documented. Approximately 30,000 lines of code are represented by the authentication library, and the rest by IMAP. The IMAP codebase includes various compiled-in helper libraries, such as Unicode handlers, support for various mail-relevant RFCs, and support for the Maildir format. The daemon code itself is divided into: a network listener which provides the functionality of inetd and tcpwrappers in a Courier-specific fashion (couriertcpd), a binary designed for handling unauthenticated IMAP connections (imaplogin), and a binary which should only ever be run by authenticated IMAP connections (imapd).

In addition, Courier-IMAP requires its own syslog frontend (courierlogger), several libraries related to authlib, and a mandatory authentication daemon (authdaemond), which listens on a socket. (By contrast to Cyrus, Courier requires authdaemond even if UNIX passwords are not used for authentication.)

The Courier server has as a design goal protecting the system from its own end users (legitimate or otherwise). Therefore, it has a number of configuration settings related to limiting the system resources available to IMAP users.

Impressions

Courier-IMAP has many positive security-relevant features, and, at an overview level, seems very well designed. The code is relatively minimal in many important ways – no NNTP client or anonymous access is provided. Privilege separation is cleanly implemented at a conceptual level. For instance, Courier requires a separate configuration file to run imapd on its non-SSL

port and on its SSL port, so it is very easy to configure only the secure port with no risk of accidentally enabling insecure access. Also, the login design, in which the pre-authenticated IMAP connection is handled by an entirely separate binary from the one which provides logged-in functions, is very clean.

Upon closer inspection, however, many things in the code disappoint. To increase efficiency, the imaplogin process does not always terminate on unsuccessful connections. The way in which this is implemented potentially opens opportunities for the connection to be left in a partially logged-in state. A bug in this portion of the code might allow an attacker to circumvent the privilege separation. In addition, the code layout itself is hard to follow and inconsistently modular.

Compilation and configuration were difficult and seemingly broken – configure is run once for each subdirectory of the compile directory (rather than maintaining a cache of, for instance, the location of gcc, or the size of an integer), leading to slow and repetitive compilation. More seriously, the compile-time configuration takes some values from outside the configuration directory in an undocumented manner. Running make clean in the Courier directory removes files which are required for compilation, so it is necessary to restore from the original tarball in order to redo a compilation. None of these items is relevant to the running system, per se, and each may be attributable to the recent imap/authlib code split, but they are inconvenient and do not inspire confidence in the package design.

In addition, there are some features which do not seem like good ideas. The cyrus-sasl implementation which allows non-daemon authentication mechanisms seems preferable to the courier-authlib implementation which always requires a daemon, thereby mandating that the system be open to attacks against the daemon itself. The server also provides some strange user features. Most notable among these are: INBOX.Outbox, a special user-visible mailbox into which users can place messages for immediate transmission via SMTP, and loginexec, a file within a user's top-level Maildir which is always executed and removed, regardless of ownership, if it exists.

Summary of Software Observation Results

It is certainly the case that manual software observation gives an incomplete impression of software security. The overall code layout, modularity, and privilege separation are obscured from the time-constrained examiner, who sees only the set of files installed and the modularity of the main() loop.

The most visible features are this high-level view of the code, the ease of installation and compilation, and the presence and quality of documentation (particularly the subset used to assist the initial installation). It is easy to find the set of mandatory requirements and dependencies of each package in the chosen configuration, but impossible to tell what additional requirements other configurations might impose.

In addition, any extra features which are offered to the user or provided in the software's back end are very visible. If such features appear to be security risks, their effect on the examiner's opinion of the code may be disproportionate to their effect on the actual code security. In this particular case, my views on the codebases were strongly affected by the defensive attitude towards security concerns taken by the UW-IMAP documentation, and by the presence of strange features in the Courier-IMAP codebase.

Based on manual software observation alone, Cyrus appeared likely to be the most secure and UW likely to be the least secure of the three packages.

Entry/Exit Point Analysis

In this section of the paper, I return to the formal definition of attack surfaces, and describe steps taken towards a methodical measurement of the attack surfaces of my testbed IMAP servers. I report on the partial results obtained from my analysis, and discuss how the results compare to those derived by manual examination. This comparison is useful both because of what it says about IMAP servers per se, and because, in order to use entry/exit point analysis to measure attack surfaces, we need to ensure that the results obtained from that analysis are meaningful.

In addition, my analysis is also interesting in terms of the obstacles to automated measurement which I encountered. The problems encountered in this analysis, some technical and some related to the analysis methodology, need to be solved in order to meaningfully automate the measurement of the attack surface of a given codebase.

Entry/Exit Background

The entry/exit methodology endeavors to provide two major developments in the measurement of attack surfaces. The first is the ability to automate the discovery of attack surface elements. The second is the ability to numerically compare the attack surfaces of different codebases.

Automated Discovery of Attack Surface Elements

Entry/exit analysis utilizes the following simple insight into attack surface measurement: In order to launch an attack on a system, an attacker must either transmit data into the system or receive data from the system [13].

Suppose we want to look at a codebase and, at the code level, find all the places which might be part of the attack surface. Any place in the code which is part of the attack surface must contain a call which receives data from or transmits data to the outside of the system. Therefore, if we find all such points, then we have found all positions on the attack surface, and we need only verify that each such point is actually an

attack surface element, and classify each one into an attack class (a set of attack surface elements which pose the same level of risk to the system).

This opens the door for automated attack surface measurement, since it should be possible to find entry and exit points in a manner which is at least somewhat automated.

Numerical Comparison of Attack Surfaces

One major difficulty with the attack surfaces discovered by a more ad hoc analysis is that they cannot be compared to one another, because there is no clear way to numerically order the features measured. However, if we use entry and exit points to find positions within the code at which attack surface elements occur, our task becomes easier.

Suppose that, for each position in the code, we can determine what access rights are needed to run that code, and what privileges the running code has. Then, we have a one-dimensional set of access rights which are directly comparable, and a similar set of privileges. Moreover, the relative ordering we should use on these elements is clear: it is more difficult to gain administrator access than it is to gain user access, and it is more difficult to gain user access than it is to gain unauthenticated access. Similarly, root is more powerful than an imapd system user, which is more powerful than a normal user, which is more powerful than a service account.

We can then assign to privilege and access levels numerical values which are consistent with this ordering. Given these, we compute the attackability of a given attack surface item by observing that higher privilege items are more desirable targets (more attackable), and that higher access-right items are more difficult to target (less attackable). If we have defined functions

$$p : \{\text{privileges}\} \to \Re$$

and

$$ac : \{\text{access\_rights}\} \to \Re\ ,$$

and if a given item has privilege Q and access rights B, then we can define the attackability of that item as

$$\text{attackability} = \frac{p(Q)}{ac(B)}.$$[5]

Therefore, if we have two different items with comparable privilege and access levels, we can numerically compare them in terms of attackability, and we can sum over all the attackabilities in a given codebase's attack surface, and compare the sum to that of another codebase.

However, we cannot actually measure attackability as a one-dimensional quantity. This is because the attack surface, as noted in the introductory sections, consists of three different types of elements – methods, data, and channels – and there is no straightforward way to correlate elements of these three types. Therefore, entry/exit analysis should ideally yield a three-dimensional vector describing the system attackability, data attackability, and channel attackability of a codebase.

[5]This definition is arbitrary, and is chosen because it is a simple function which varies directly with privilege and inversely with access rights.

Methodology for Entry/Exit Point Measurement

In this section, I discuss both the methodology I used for the successful measurement of method entry and exit points in the IMAP codebases, and my efforts to measure data and channel entry and exit points.

Measuring Data and Channels

The insight in measurement of data and channels is that, because of the way UNIX is designed, a process cannot read or write to a channel or persistent data object without making either a system call or a call to an external library which makes that system call itself.

Of course, we do not know the identities of all those external system or library calls. However, they are easy to find: a binary examination tool such as `nm` [8] will find the list of symbols which are referenced by a piece of compiled code. Then, a source code examination tool such as `ctags` [6] will give us a list of symbols actually defined within the code. Any symbols which show up in the `nm` output but not in the `ctags` output must be externally defined.

Once discovered, these external methods must be examined manually to determine whether they access data or channels, and what manner of access they perform (read, write, create, delete). However, in my experience, there were only about 150-200 unique external methods per codebase. Even better, these methods are provided by external libraries or by the operating system, rather than by the examined codebases. Therefore we can expect them to behave identically across codebases. Once the behavior of `fopen()` has been classified on a given operating system for one codebase, that classification can be used without modification for any other codebase running on that operating system. I was able to classify the calls made by my sample codebases, and to obtain a list of internal methods which made calls to external data and/or channels.

From there, the task became more difficult. The UW-IMAP codebase contained 896 instances in which an internal function made an external call involving data or channel access. That number might correspond to even more actual references. (For instance, a modular codebase might define an internal function whose job was to open a file and read from it. That function might be referenced from multiple different places in the code, each time with a different filename.) A very good code analyser might be able to find internal object names, leaving us with statements of the form "routine *X* read from the file whose name is stored in the variable `foo` at line 796." However, I did not have access to a code analyser of that quality, and, even so, this tells us nothing about which entities outside of the given codebase require permission to read or write the file, nor about whether the codebase's installation

procedure faithfully adheres to the strictest permissions possible rather than negligently substituting more relaxed permissions than needed.

We can instead work from the other end, and use software observation in a jailed environment to obtain the full list of files that each codebase requires in order to operate. However, this tells us only about the files required by the specific configuration used in the example jail, rather than the full set of files which might be required or used by the software.

Thus, using entry and exit points to count data and channel attack surface elements is not yet a solved problem.

Measuring Methods

This problem is more tractable. We would like to measure the set of all internal code which is runnable by an attacker. Provisionally, we measure code at the function level. First, we look at code which the attacker can call directly. Each of the three IMAP daemons uses a function with a name like main() to receive all input from the remote user, meaning that each daemon provides one reachable function. This is not very interesting.

Therefore, we can extend our analysis by looking at functions which are themselves called by those directly reachable functions, and are thus indirectly reachable by the attacker. We obtain these function names using a program flow analysis tool. In the case of IMAP analysis, I used a tool called cflow [7]. Display 1 shows some sample cflow output from the UW-IMAP case.

This gave several hundred lines of output per codebase, the name of every function reachable from the main() function. In order to count each method in the attack surface, I needed to find out what access rights it had, and with what privilege it ran. Some manual preparation was needed in order to obtain that information from the codebase.

On a UNIX system, the privilege with which the code runs is the privilege with which it started, unless a system call has occurred which changes privilege, such as setuid(). If a program starts running as root and later drops privilege, then any functions called before the setuid() call are called as root, and any functions called after setuid() are called as an unprivileged user.

It is slightly more difficult to determine the access needed to reach a certain element of the code, because it is necessary to find each code location at which authentication is performed. For instance, a call which compared the password provided by the user to one retrieved from a local file would constitute a change of authentication state – before that comparison, a user is unauthenticated, but after the comparison is successful, he is authenticated.

However the access is determined, once we know where the access and privilege changes occur, we can edit the directly reachable routine, creating a copy for each access and privilege pair which contains only those calls reachable from that level. Here is an example from my edit of the Cyrus codebase. Display 2 shows the original code fragment, from the part of the main loop which handles the AUTHENTICATE command.

In Cyrus, the variable imapd_userid is always set upon successful authentication, and at no other time. I edited the code fragment shown in Display 3 with that in mind, retaining only the code accessible to an unauthenticated user.

Note that I needed to verify that it is possible for authenticate_command() to return to the main loop upon failure. If authenticate_command() instead killed the process, or jumped to another segment of code, then I would have considered the call to snmp_increment() to also be unreachable by an unauthenticated user, and would have deleted it as well.

Once the edited copy had been made, I could then rerun cflow and obtain a subset of the original code elements, the subset reachable by the unauthenticated user. I then repeated this process for the Cyrus authenticated user, for the anonymous user, and for the administrative user, and for all combinations of access and privilege in each of the UW-IMAP and Courier codebases.

Obstacles to Entry/Exit Point Measurement

In the process of counting reachable methods, I encountered a number of obstacles, some technological, some integral to the enumeration process. I mention both types of difficulties, since technological problems might be solved by better tools, but, if those tools do not exist, the process of writing them offsets the time gains of automation.

Determining Whether Data Has Been Transmitted

In principle, we are not concerned with all functions reachable from the direct entry point, but only with functions through which the attacker can transmit the data required by his attack. How do we measure the subset of functions which actually allow the attacker to transmit data?

Clearly, if an attacker can input arbitrary data which is immediately read into an internal buffer belonging to the function, he has transmitted data into

```
1  main {src/imapd/imapd.c 253}
2    strrchr {}
3    mail_parameters {src/c-client/mail.c 297}
4      fatal {}
5      env_parameters {src/osdep/unix/env_unix.c 150}
6        fs_give {src/osdep/unix/fs_unix.c 57}
7         ... mail_parameters ... {3}
8         free {}
```

Display 1: Partial cflow output from UW-IMAP codebase.

the function. However, consider a more restrictive case, in which, for instance, the input must have a certain format, such as being a valid SSL key, but is otherwise chosen by the attacker. Then consider the case in which the attacker can only insert a single integer value, such as a return code. Consider the case in which the attacker's return code is ignored by the calling function.

My conclusion was that the safest route is to count every called function, regardless of the syntax or content of the call. Display 4 shows an example which supports that conclusion. It is taken from the UW imap-2004a codebase[7] which was reported to be vulnerable in early 2005 [10] (all code irrelevant to the attack has been excised for clarity).

The problem here is caused by the logic used to set the variable u – the check done on md5try is backwards, so that, if md5try is 0 (if the attacker has tried to authenticate too many times using CRAM-MD5), the call automatically succeeds where it should automatically fail. The attacker does not need to send any particular exploit data in order to break into this system. All he needs to do is to attempt CRAM-MD5 authentication unsuccessfully three times, and then he will be authenticated on the fourth try.

In that light, it seems very reasonable to claim that causing a line of code to be run is sufficient, from an entry/exit point perspective, to transmit data into that code.

Undercounts and Overcounts

In order to automatically enumerate reachable methods, it is necessary to have an accurate method of detecting code execution paths. There are many tools which profess to do this, but, in practice, there are problems which lead to undercounts or overcounts.

First, in order to avoid undercounts, it is necessary to have a program which parses the code accurately. I used cflow for my final analysis because it employs gcc -E, and therefore works quite well.

[6] Cyrus-IMAP is copyright 1994-2000 Carnegie Mellon University. Some function and variable names have been modified at the request of CMU.

[7] UW-IMAP is copyright 1988-2004 University of Washington.

```
  ...
  if (imapd_userid) {
    protocol_printf(imapd_out,
      "%s BAD Already authenticated==>[ignored: r]<====>[ignored: n]<==", tag.s);
    continue;
  }
  authenticate_command(tag.s, arg1.s, haveinitresp ? arg2.s : NULL);
  snmp_increment(AUTHENTICATE_COUNT, 1);
} else if (!imapd_userid) goto nologin;
else if (!strcmp(command.s, "Append")) {
  if (c != ' ') goto argsmissing;
  ...
```

Display 2: Original Cyrus-IMAP authentication code fragment.[6]

```
  ...
  if (imapd_userid) {

  }
  authenticate_command(tag.s, arg1.s, haveinitresp ? arg2.s : NULL);
  snmp_increment(AUTHENTICATE_COUNT, 1);
} else if (!imapd_userid) goto nologin;
  ...
```

Display 3: Modified Cyrus-IMAP code: subset of code accessible to an unauthenticated user.

```
static int md5try = 3;

char *auth_md5_server (authresponse_t responder,int argc, char *argv[])
{
  char *ret = NIL;
  ...
  u = (md5try && strcmp (hash,hmac_md5 (cha1,c1,p,p1))) ?  NIL : user;
  ...
  if (u && authserver_login (u,authuser,argc,argv))
    ret = myusername ();
  else if (md5try) --md5try;
  ...
  if (!ret) sleep (3);    /* slow down possible cracker */
  return ret;
}
```

Display 4: Vulnerable CRAM-MD5 function in UW imap-2004a.

However, cflow ignores function arguments and program logic, which one might prefer to take into account.

For instance, in the following example, method_c is not actually reachable from method_a, but cflow would claim it was:

```
method_a() {
  method_b(DONT_USE_X);
}

method_b(int flag) {
  if (flag != DONT_USE_X) {
    method_c();
  }
}
```

Other tools which perform some of these functions include doxygen [5], which parses function arguments in a relatively usable fashion, but has a poor understanding of C syntax and does not deal with logic, and cqual [3], which is intended to handle both function arguments and program logic, but is difficult to work with and possibly buggy in practice.

A major source of undercounts in the IMAP case is caused by function pointers – variables which store the names of functions. At the point of execution, the function is referenced only by the name of the variable, so there is no reliable way to tell what actual function is being executed. All three IMAP codebases make use of function pointers, and no tool I found was capable of parsing them.

Classifying Multiply-Accessed Methods

Once the codebase has been divided by privilege and access levels, many functions are identified which are accessible by code running at different levels. This code may belong to internal or external helper routines, or it may be authentication-relevant code which has inadvertently been left too open. In order to classify the code, I needed to generate a strategy for handling such functions.

One strategy would be to count each function twice, once per privilege/access pair from which it was accessible. However, this seemed to overcount in the case of, for instance, an administrative user and a normal user who could both access normal user code.

Another approach would be to count only the worse of the two levels, i.e., if the function was accessible to either an unauthenticated user or to an authenticated user, count it as unauthenticated. However, this would probably provide an undercount, because an authenticated user might be able to reach more of the functionality of a routine than an unauthenticated user. If there are two attack paths related to a given function, both should be counted.

I compromised by creating new privilege and access levels to reflect multiply-accessible code. For instance, if root and user are privilege levels, then root+user would be the privilege level for code which could be reached at either of those levels. In the IMAP codebases, the appropriate total orderings of privilege and access levels were clear even with the additional levels.

Results of Entry/Exit Point Measurement

For each codebase, I identified the function responsible for receiving network connections from remote users, and broke down the codebase by privilege and access levels as previously described. For the UW and Cyrus codebases, this was a simple matter of finding the single input-handling function.

The Courier codebase was somewhat more complex. In that case, the program couriertcpd receives and processes an incoming network connection, does some checking, and then hands the connection off to imaplogin, which processes authentication routines. If imaplogin reports success, then an imapd process is spawned. Therefore, in order to find all the routines which could run at each privilege and each access level in the Courier case, I needed to analyse all three of these programs.

I also performed an analysis of Courier in which I omitted the couriertcpd code, which all runs as root and is accessible to unauthenticated users. Since the Courier and Cyrus codebases include network listeners in their codebases, while UW relies on inetd to provide network connectivity, I was interested in gauging the amount of code required to listen for network connections.

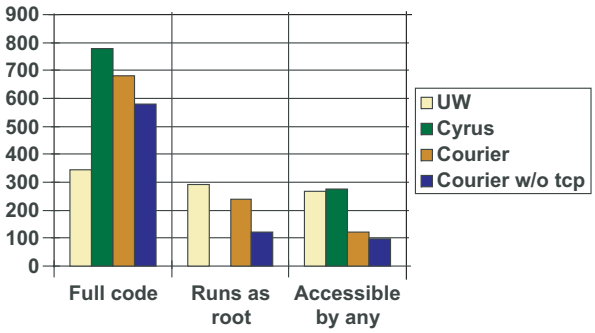

Figure 2: Number of reachable methods in each codebase: (a) full codebase, (b) code which runs as root, (c) code accessible at any access level.

Method Counts By Codebase

Figure 2 shows a representative subset of breakdowns of method counts among the codebases. The left graph shows the total number of methods reachable in each codebase. The middle graph shows the number of methods reachable by code which is running as root. (Note that Cyrus has no reachable code which runs as root.) The right graph shows the number of methods which are open to all access levels defined for the codebase. That is, in the UW case, this graphs functions which are reachable by an unauthenticated user, by an anonymous user, and by an authenticated user. In the Cyrus case, this graphs functions which the unauthenticated, anonymous, authenticated, and administrative users can all reach. In the Courier case, this graphs functions which the unauthenticated and authenticated users can both reach.

Total Orderings for Privilege and Access Rights

Once I had counted all reachable code methods, I needed to populate the total orderings for the privileges and access rights. All types in each ordering needed to be assigned numerical values, so that the attackability of each attack class, and thus of the entire system attack surface, could be computed.

Access Rights	Points
admin	8
auth	4
anon	1.5
auth + anon	1.45
admin + auth + anon	1.4
unauth	1
admin + unauth	0.95
auth + unauth	0.9
admin + anon + unauth	0.85
auth + anon + unauth	0.8
admin + auth + anon + unauth	0.75

Table 1: Attackability ordering for method access rights (higher value is harder to attack).

Privilege	Points
service	2
user	3
user + service	4
imapd	7
root	10
root + user	13
root + user + service	14

Table 2: Attackability ordering for method privileges (higher value is more valuable target).

Table 1 contains the ordering I derived for access rights, and Table 2 contains the ordering for privileges. Note that the ordering results naturally from known information about security of UNIX servers and the IMAP domain, but that the specific numerical values chosen are arbitrary.

System Attackability of IMAP Servers

Table 3 shows the system attackability of each IMAP server given these values and the reachable methods determined by the automated method. This table shows that, according to the attackability metric used here, Courier is the least vulnerable of the servers, while UW and Cyrus score similarly. The results also indicate that much of Courier's vulnerability is caused by its network listening code. This implies that Cyrus, which also contains network listening code, might be less vulnerable than UW were the functionality provided to be taken into account.

However, Figure 3 is worth noting. It demonstrates that the attackability metric is somewhat dependent on the exact numbers of points chosen for various privileges and access rights, even if the total ordering is held constant. By default, I assigned seven points to the special imapd user employed by Cyrus. However, a site which was most conscious of the need to avoid root compromise on their machines could assign imapd five points, close to the value for an unprivileged user, and would find that Cyrus significantly outperformed UW in attackability. A site with a different posture could note that any attack on the imapd user risked opening up all users' e-mail, and therefore was almost as bad as a root compromise. If this site assigned nine points to imapd, Cyrus would suddenly appear radically more attackable than UW.

Codebase	System Attackability
UW-IMAP	5044
Cyrus	5217
Courier	3122
Courier w/o tcp	1813

Table 3: System attackability of IMAP servers.

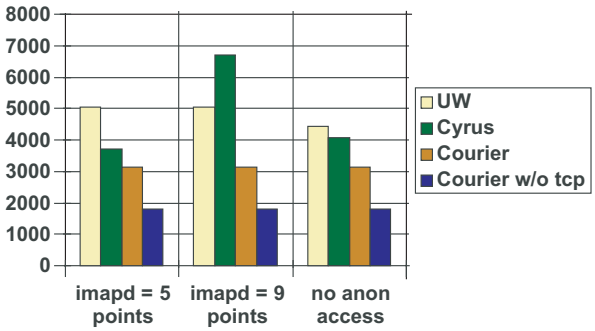

Figure 3: System Attackability of IMAP servers, with (a) imapd user worth 5 points, (b) imapd user worth 9 points, (c) imapd user worth 7 points and no anonymous access.

This tradeoff between protecting root and protecting user mail from other users is symptomatic of an obvious security consideration which arises when trying to decide whether to use Cyrus or a more conventionally-designed IMAP server. This example is included to demonstrate that the attackability metric does not protect us from having to make that judgment call.

In general, however, the attackability metric seems to agree reasonably well with observation. Despite the large size of the Cyrus codebase, its attackability is similar to that of UW-IMAP, indicating that Cyrus has good privilege separation while UW-IMAP does not. Indeed, the major effect of this metric as applied is to reward code with good privilege separation, which is a reasonable goal, though only one of many.

The `couriertcpd` example demonstrates that it is difficult to compare systems which perform different

functions, and that even between systems which are very similar in functionality, one system may provide many more of its own dependencies. In the process of measuring attackability, it is necessary to consider the external systems on which each codebase depends.

Conclusions

In this section, I discuss the feasibility of performing an automated entry/exit analysis as a means of comparing software security. I also discuss future work needed to determine whether entry/exit analysis of attack surfaces will be practical, and whether it will be useful. I close with a few words about the security of the specific IMAP servers I studied.

Feasibility of Entry/Exit Analysis

One important point is that it is difficult to ensure that any analysis is comprehensive. In the particular case of this analysis, I looked at only the code available via the network API, and did not examine the code added by periodic cleanup processes, LMTP, etc. I also did not look at the authentication daemons provided by Cyrus and Courier as part of the authentication facility. IMAP communicates with those via socket, so they would have surfaced had I done a channel analysis. It is still necessary to manually examine the code's functionality in some detail in order to find starting points for automated analysis.

In addition, entry/exit analysis is not trivial to perform. Approximately four to eight hours of labor was required to prepare each codebase for analysis by cflow, and that took into account significant familiarity with the code in question. The most time-consuming aspect is dividing the code based on privilege and access levels, but some time is also required to find the correct internal function definitions within a complex codebase, given that cflow and related tools cannot read Makefiles.

The division of the code is not entirely automatable work – some judgment calls are required in order to determine, for instance, at exactly what point authentication takes place. In theory, the Courier approach, in which a different physical piece of software handles execution at each privilege and access level, should make things easier. In practice, there are thorny issues even in that implementation, since authentication, setuid(), and exec() of the other daemon do not all take place simultaneously, nor even necessarily within the same function as one another.

In addition to all this, I did not attempt the data or channel enumerations, which would be more complex and difficult. In those cases, it might be complicated to even determine the total ordering among access rights – is a channel which can restrict by IP addresses higher or lower risk than a channel which implements anti-DoS protection?

However, entry/exit analysis holds the promise of giving usable information which speaks meaningfully about the internal security posture of a codebase, and which is derived from less work than that required for a comprehensive manual analysis.

Future Work

Measurement of system attack surfaces could be improved by viewing the code at a level other than the function level, i.e., by looking inside functions and giving more weight to larger or more complex ones in analysis. This would be aided by the use of better tools which could trace the flow of execution through the code more accurately. It would also be useful to take into account which functions were actually compiled into the code in use, perhaps by comparing the full reachable code tree with symbol analysis of the resulting binary, or by replacing cflow with some tool which actually could read Makefiles.

Performing a data and channel analysis of entry and exit points would also provide a significant amount of information about the practicality of using entry/exit analysis to obtain a full picture of the system.

In addition, the problem of comparing codebases which perform the same overall function, but not the same subsets of that function, will need to be solved. For instance, if it were possible to determine what subset of the Cyrus and Courier codebases was dedicated to providing port listener functionality, then administrators could compare an analysis of inetd to just those subsets, and thereby determine whether the Cyrus and Courier implementations of that functionality were more or less risky than a third-party implementation.

Comparison of IMAP Server Software

I will finish with some brief words about the relative security of IMAP servers as a result of this analysis. My subjective impression that the Cyrus and Courier codebases are better designed than the UW codebase was corroborated by the entry/exit analysis results. In particular, the lack of internal privilege separation in UW-IMAP indicated that that codebase is not designed in a security-conscious way. In comparing Cyrus and Courier, I determine that Cyrus is better built, but overreaches significantly in terms of the number and diversity of features it offers, while Courier is designed in a very security-conscious manner, but implemented somewhat more strangely. My conclusion is that despite the administrative headaches it introduces, Courier is likely to be the best security risk when choosing one of these three products to act as an open source IMAP server.

Acknowledgements

I would like to thank Jeannette Wing and Dawn Song for enabling me to do this project, introducing me to the attack surfaces framework, and giving me advice and assistance along the way. I would also like to thank Pratyusa Manadhata for all his help with attack surfaces and entry/exit points, and Rob Siemborski for his explanations and corrections on the subject of IMAP implementations.

Availability

Because much of my entry/exit analysis was performed manually, there is no distributable software associated with this project. The project page at http://www.glassonion.org/projects/imap-attack/ contains a more detailed description of the methodology, along with scripts I used, and the detailed output of the analysis.

Author Information

Chaos Golubitsky first worked in system administration while an undergraduate at Swarthmore College. After earning a B.A. in Mathematics and Computer Science, she worked at the Harvard-Smithsonian Center for Astrophysics for three years, where she became interested in log monitoring. She returned to school for an M.S. in Information Security from Carnegie Mellon University, where she focused on practical analysis and improvement of system and software security.

References

[1] *Common Vulnerabilities and Exposures Database*, http://www.cve.mitre.org .

[2] *Courier-IMAP*, http://www.courier-mta.org/imap/ .

[3] *Cqual*, http://www.cs.umd.edu/~jfoster/cqual/ .

[4] *Cyrus IMAP Server*, http://asg.web.cmu.edu/cyrus/ .

[5] *Doxygen*, http://www.stack.nl/~dimitri/doxygen/ .

[6] *Exuberant Ctags*, http://ctags.sourceforge.net/ .

[7] *FreeBSD Ports: cflow*, http://www.freebsd.org/cgi/url.cgi?ports/devel/cflow/pkg-descr .

[8] *GNU Binutils*, http://www.gnu.org/software/binutils/ .

[9] *University of Washington IMAP*, http://www.washington.edu/imap/ .

[10] "UW-imapd fails to properly authenticate users when using CRAM-MD5," *Vulnerability Note VU#702777*, US-CERT, http://www.kb.cert.org/vuls/id/702777 , January, 2005.

[11] Howard, M., J. Pincus, and J. M. Wing, "Measuring Relative Attack Surfaces," *Proceedings of Workshop on Advanced Developments in Software and Systems Security*, August, 2003.

[12] Kamp, P.-H. and R. N. Watson, *Jails: Confining the Omnipotent Root*, Technical report, FreeBSD Project, http://docs.freebsd.org/44doc/papers/jail/jail.html .

[13] Manadhata, P., "Entry Points and Exit Points," Personal communication, 2005.

[14] Manadhata, P. and J. M. Wing, *Measuring a System's Attack Surface,* CMU-CS 04-102, Carnegie Mellon University, January, 2004.

[15] Mullet, D. and K. Mullet, *Managing IMAP*, O'Reilly Media, Inc., 2000.