Measuring Attack Surfaces of Open Source IMAP Servers

E. Chaos Golubitsky

May 2005

Abstract

The attack surface metric 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. The attack surface metric can be used to compare the security of multiple systems which provide the same function.

The Internet Message Access Protocol (IMAP) is a protocol which has been in existence for over a decade. Relative to web (HTTP) and e-mail transfer (SMTP) servers, IMAP servers are a niche product, but they 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 project, i use the attack surface metric to discuss 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.

Contents

1 Introduction

System administrators and others working in operational contexts must 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 avoid known vulnerabilities by installing the most up-to-date version of the software, and by configuring it sensibly. However, some software is more prone to future attacks than others. Therefore, those responsible for selection of software packages benefit from a metric which can provide 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.

The attack surface of a software package is one such 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. In this paper, i describe steps taken towards performing a relative vulnerability assessment of open source Unix IMAP servers based on measurement of attack surfaces.

1.1 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 obtained in this manner to those obtained via an ad-hoc analysis.

In the remainder of section 1, 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. In section 2, i explore the high-level interactions of IMAP servers with their environments. Then i introduce the three target IMAP servers specifically and discuss the design philosophies and implementation peculiarities of each. In section 3, i perform an ad-hoc manual analysis of IMAP server attack surfaces, in order to provide a baseline with which to compare the automated analysis. In section 4, 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 stumbling blocks to performance of a fully-automated analysis. In section 5, i compare the results of the analyses performed, and provide conclusions and future directions for attack surface research.

1.2 Attack Surfaces

1.2.1 Theory of Attack Surfaces

The attack surface metric is a comparative software vulnerability metric for packages which perform similar functions. The 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 some access to which he is not entitled. In order for attack to be possible, the system must provide some action to which the attacker has access before his attack has succeeded. In order for attack to be desirable, 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 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 gain 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 interprocess communication (IPC) mechanisms potentially usable by an attacker, and (3) data items, which are sources of persistant data (like 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 cryptographically authenticate themselves before communicating further may pose significantly 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.

1.2.2 Prior Work on Attack Surface Measurement

The idea of attack surfaces was introduced by Howard in 2003, and was used to measure the relative attack surfaces of versions of Windows. [12] Manadhata and Wing extended the description of attack surfaces in 2004, and discussed ways of measuring attack surfaces in the context of Linux versions. [15] Their analysis introduced the concepts of types and of attack classes. Types are a way of extending actions and resources by taking into account security-relevant attributes of those items. So, if "http daemon" is a method, then "http daemon running as root" could be an extended method, and if "file" is a data item, then "file writable by group `users`" could be an extended data item.

An attack class is a set of methods or resources which share the same type, and therefore the same

attackability. Once attack classes are identified and their members within the systems enumerated, it is possible to compare systems by counting elements of dangerous attack classes.

Wing and Manadhata used the Common Vulnerabilities and Exposures (CVE) database [2] to identify dangerous attack classes. They used CVE entries related to versions of the Linux operating system to identify 14 resource types which were most prevalent in attacks on Linux. 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, identifying attack classes based on previously-discovered vulnerabilities makes the assumption 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 section 4 of this paper attempts to address these issues.

1.3 IMAP Servers

The Internet Message Access Protocol (IMAP) is a protocol which provides e-mail access to remote authenticated users. In the standard e-mail model, messages are passed from site to site by Mail Transfer Agents (MTAs), each of which examines the message to see if its recipient is local, or if it needs to be forwarded to another site. Once an MTA recognizes the recipient as local, it must place the 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 Berkeley mbox or Maildir. 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 like SMTP or LMTP. 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. [16]

IMAP is an interesting target for this security analysis for several reasons:

- IMAP is an open protocol[1] with known requirements, so it is possible to analyse the desired behaviour of the code systematically, rather than treating it as a black box.
- There are three freely available open source IMAP servers which share the bulk of the market

[1]The version of the protocol with which my three reference servers comply, called IMAP4rev1, is specified by RFC 3501.

— UW-IMAP, Cyrus, and Courier-IMAP. Each server has been in development for several years, so the codebases are likely to be fairly mature.

- The codebases which make up IMAP servers have sizes between 100,000 and 250,000 lines of code. This is large enough that it is not practical to perform an analysis by reading every line of the code, but small enough that it should be possible to get a design-level sense of the actions the code is taking.
- Although the IMAP protocol has been in existence since the late 1980s, widespread use of IMAP has happened much more recently than widespread use of protocols like SMTP or HTTP. IMAP's popularity has been due in large measure to interest in reading corporate or ISP e-mail over the web, which has really only occurred within the past five years. Thus, there has not been a systematic security analysis of any sort related to IMAP servers.

My goal was therefore to study the attack surfaces of the three popular IMAP implementations.

2 Observation of IMAP Server Software

In order to assess the effectiveness of the attack surface analyses, i needed to gain a subjective impression of the relative security postures of the three IMAP server packages. I performed this assessment by installing each server package in a constrained environment. To the extent possible, i tried to apply the same minimal configuration to each server.

2.1 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. [13]

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.

The preferred configuration for each IMAP server was a minimal default configuration in which the server listened only on port 143 for both encrypted and unencrypted connections[2], and in which

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

there was one user account which could be authenticated via a CRAM-MD5 database.

This installation had two major purposes. One was to provide a testbed for observing the software's behaviour during the performance of other analyses. The second was to enable me to obtain a subjective baseline of the software's security. In particular, i was able to assess each package's installed size and dependencies, behaviour during compilation and configuration, network and system behaviour during operation, and any notable extra features the software provided, be these user-visible options, extra daemons and port listeners, or API methods.

In the process of installing the software, i also obtained all accessible previous versions of the codebases for each of the software packages. In each of the three cases, it was possible to find versions going back several years. This allowed me to track differences between code versions over time, and to more easily examine and classify vulnerabilities in older versions.

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 (where relevant), and my subjective impression of the security of the package.

2.2 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 is a graphical representation of the interactions described in the remainder of this section.

2.2.1 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, primarily specified by the IMAP RFC. These commands are sent by the user to the server using a fixed format and with a command-specific number of arguments, and have an RFC-defined effect. The command set includes things like `CAPABILITY` (list the capabilities offered by this IMAP server), `STARTTLS` (negotiate encryption for this connection), `AUTHENTICATE` (perform some sort 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]

2.2.2 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 receipt 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 must share disk access with the IMAP server, and, 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.

2.2.3 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 are 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 like `inetd` to launch the IMAP server in response to connection requests.

Third, each IMAP server requires some non-standard libraries for various reasons. 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.

2.2.4 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

[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.

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.

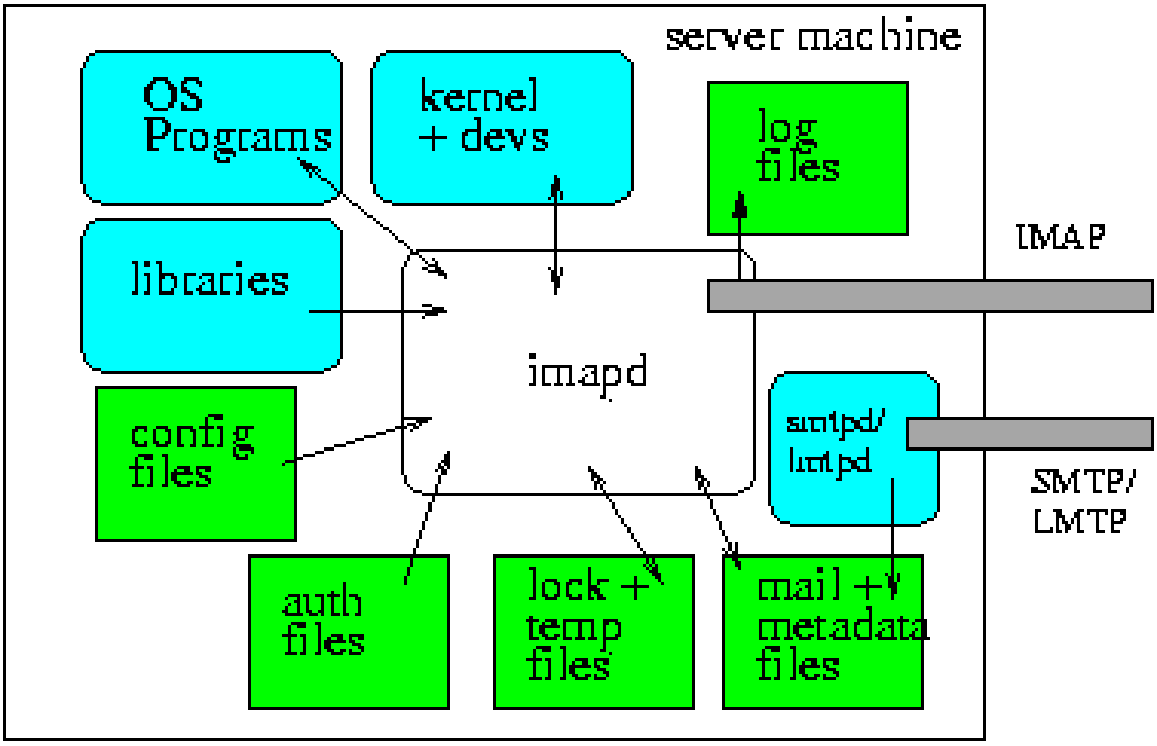

Figure 1: High-level IMAP server interactions and dependencies (executables in blue, files in green, imapd may provide lmtpd service internally)

2.3 UW-IMAP Server

2.3.1 Overview

UW-IMAP [10] 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 developed 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, which 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. Then, an `imapd` process is spawned, handles a single IMAP connection, and 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, as well as IMAP, POP3, and NNTP types to allow the server to act as a client for folders served by an unrelated third site. 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.

2.3.2 Impressions

The UW codebase does not have a good security history, and the design does not inspire confidence about its future. Its benefits are 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 relatively small API set is due to the fact that the UW author wrote the IMAP RFC, which defines the minimal allowable set of API functions. Therefore UW-IMAP can provide only RFC-required functions, while other servers must provide both the RFC functions and any additional functions desired by those servers.

However, the drawbacks are many, and seem to go down to the design philosophy of the package. The code is not modular — as an example, the `imapd` program's `main()` 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, i am concerned that functions may be compiled in to the UW server which are really only necessary or desirable for client operation.

The design decision that IMAP access is only one method into a user-accessible system is also worrisome. First off, it effectively prevents administrators from building a closed box IMAP server.[4] Secondly, this design encourages other strange features. Most notably, 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. This is not a vulnerability in itself, but it is strange nonetheless.

Despite UW'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 author makes it clear in the FAQ that he considers replacement of such calls with their length-checking counterparts to be undesirable. The author also expresses his opposition to GNU configure, preferring to implement his own custom makefiles. This choice is not necessarily related to security, but it causes code compilation to be less uniform and predictable 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 — with officially unpredictable results.

2.4 Cyrus Server

2.4.1 Overview

Cyrus is written and maintained by Project Cyrus at Carnegie Mellon University. [5] 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 internally-defined 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 developed 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, provided 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` is

[4] 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.

in itself 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 process `master` which listens for incoming network connections and passes 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 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[5], is provided for enduser 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 endusers to edit their filters — the most common is an extra port listener through which users can authenticate and make changes over the network.

2.4.2 Impressions

I found Cyrus to be the most impressive of the codebases in terms of code 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

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

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 (like 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.

2.5 Courier-IMAP Server

2.5.1 Overview

Courier-IMAP is the IMAP server component of the Courier MTA. [3] The IMAP server is packaged with the MTA, but is also available and configurable as a standalone 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 courier/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 socket-listening authentication daemon (`authdaemond`). (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 endusers (legitimate or otherwise). Therefore, it has a number of configuration settings related to limiting the number of system resources available to IMAP users.

2.5.2 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 are provided. Privilege separation is cleanly implemented at a conceptual level. For instance, Courier requires separate configuration 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 conceptually very clean.

Upon closer inspection, however, many things in the code disappoint. For efficiency purposes, the `imaplogin` process does not always terminate on unsuccessful connections. The way in which this is implemented potentially leaves open 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` and the size of an integer), leading to unnecessarily slow and repetitive compilation. Also, the compile 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 by the compile, so it is necessary to restore from the original tarball in order to redo a compilation. None of these items are relevant to the running system, per se, and they may all be attributable to the recent imap/authlib code split, but they are inconvenient and do not inspire confidence in the code 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.

2.6 Summary of Software Observation Methodology and Results

It is certainly the case that manual software observation gives a biased 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 required for the initial installation). It is easy to find the set of mandatory requirements and dependencies of each package, though this only applies to the requirements of the particular configuration the examiner has chosen.

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 and dismissive attitude towards security concerns taken by the UW-IMAP documentation, and by the presence of very strange features in the Courier-IMAP codebase.

As a result, my initial ranking based on software observation would have placed Cyrus as likely to be the most secure, and UW likely to be the least secure.

3 Ad-hoc Attack Surface Analysis

I first obtained a baseline attack surface measurement by performing an ad-hoc analysis. This is similar to the type of analysis which was performed on Linux by Wing and Manadhata. However, analysis of a software package provides a different set of challenges than analysis of an operating system.

3.1 Ad-hoc Measurement Methodology

The necessary steps for measuring an attack surface are:

- Identify resources which are likely to appear on the attack surface, and which can be measured within the codebase.
- Enumerate these resources within each codebase.

- Identify a set of types which should be used to extend the resources and form attack classes, and determine which of these attack classes carry the highest risk.
- Classify the previously-identified resources based on types.
- Summarize the results, and draw conclusions about the relative attack surfaces of the code-bases.

3.2 Enumeration of IMAP Server Resources

For the Linux analysis, resources were identified based on analysis of the Common Vulnerabilities and Exposures database [2], so i first performed a survey of CVE entries related to IMAP servers.

3.2.1 CVE Analysis

Searching for "Linux" in the CVE yields many results, but "IMAP" gives significantly less data. The initial search turned up 53 CVE entries for further examination, of which 32 were IMAP server vulnerabilities. (The remainder affected IMAP clients, or programs like `inetd` or OpenSSL on which IMAP may depend.) Of the server vulnerabilities, 10 affected UW-IMAP, 7 affected Cyrus, 4 affected Courier, and the remainder affected closed source IMAP servers outside of my study.

I classified the vulnerabilities based on the type of access which was needed to trigger the attack, and on the type of target of the attack. I searched for information by looking at all the references listed in the CVE posting, searching for relevant Bugtraq [1] posts, and, where relevant, comparing code versions before and after patching. I found some further information about 30 of the 32 reported vulnerabilities.

In 22 of the remaining 30 cases, the attack was triggered by an IMAP API input (either authenticated or unauthenticated client input over the IMAP channel), the target was arbitrary code execution as either root, the imapd account, or the authenticated user account (as appropriate), and the attack was caused by a buffer overflow, typically in an argument to the API method.

The remaining 8 cases were: two in which the vulnerability could be exploited by a malformed e-mail message introduced into the system, three in which an error allowed an IMAP user to gain configuration information about the system, possibly including authentication data, one in which an authenticated IMAP user could run an illegal command, and two in which a user with shell access to the IMAP system could gain information or cause a denial of service to other users.

Compared to the Linux case, the IMAP CVE data was quite sparse. Over 70% of the vulnerabilities found were caused by a very simple exploit, namely, by sending a long argument string or long command name during the IMAP server dialogue. Given this, it seems reasonable to suspect that the surface has barely been scratched with regard to IMAP server vulnerabilities. Certainly, the CVEs

did not form a definitive record of the types of IMAP server exploits possible, so it was necessary to look to other sources of attack surface information.

3.2.2 Other Sources of Resources

In addition to the CVE analysis, i used several other sources to find IMAP server resources. First, i used the list of data items in the per-server jails to determine file dependencies for each server. The list of files obtained via jail examination is sufficient only to identify the files needed by the specific configuration i installed; a different configuration might require other files as well.

I also used the results of previous attack surface measurements to identify sets of resources which should be measured. For example, Howard's examinations of HTTP servers have indicated that it is necessary to count authentication mechanisms provided, in addition to the presence or absence of an `AUTHENTICATE` command.[6] This makes sense, since each authentication command which exists in parallel gives an unauthorized person another opportunity to break the authentication somehow.

My last source of attack surface resources was the examination of code and documentation which i had performed in search of an informal understanding of the security of each IMAP server. As a byproduct of this analysis, i found a set of surface elements which were possessed by at least one of my target servers. I then searched the documentation and code of the other servers to see whether they too contained instances of that element.

3.2.3 Resources Measured

The resources i enumerated fell into the three basic categories of methods, data, and channels. Three types of methods (executable code) were measured. First, i counted methods made available to remote users via the IMAP API. The IMAP specification requires the remote user, upon establishing a TCP connection with the IMAP server, to enter a command line of the form:

```
ID COMMAND ARGS ...
```

where *ID* is an identifying tag for the command exchange, *COMMAND* is the command to be run, and *ARGS* are optional arguments. I enumerated API methods by using the code which parses this exchange to find all valid values of *COMMAND*, each of which is an API method.

Second, i counted authentication mechanisms made available to the enduser. All authentication is accomplished using one of the commands `LOGIN` or `AUTHENTICATE`, but `AUTHENTICATE` takes an argument indicating the type of authentication desired. Therefore, counting only the two authentication-related API methods underestimates the number of possible means of authentication. However, since all three codebases implement some sort of flexible authentication, it is necessary to exercise some

[6] Michael Howard (2005), personal communication

discretion in counting mechanisms. Therefore, i counted mechanisms for which a different subset of IMAP code is used, i.e. the file `auth_gss.c` rather than `auth_md5.c` in the UW case, or a database query from `authdaemond` rather than a lookup in the password file in the Courier case.

Third, i counted internal server features which provided some extra functionality to the remote user, but were not implemented in the form of a user-end command. This category included such items as the ability for a server to act as an NNTP client, or UW's flexible mailbox type guessing, or Courier's `loginexec` feature, or extra processing which could be compiled in to handle certain types of broken IMAP clients.

	UW	**Cyrus**	**Courier**	**TOTAL**
API method	35	54	36	125
authentication mechanism	5	5	4	14
feature	6	5	4	15
TOTAL	46	64	44	154

Table 1: Counts of method resources found by ad-hoc analysis

Eight general classes of files were measured, based on the jail contents and on the diagram of IMAP server interactions in section 2. However, these classes were more usefully grouped into two broad categories. User files are those file types which are intentionally provided to the remote user in some fashion. These files include mail files, mail metadata files, and temporary files. Backend files are those file types which are to be read only by the server, and are not intended for remote access. They include configuration files, authentication files, executable files, libraries, and lockfiles.

	UW	**Cyrus**	**Courier**	**TOTAL**
backend file	18	26	36	80
user file	5	11	7	23
TOTAL	23	37	43	103

Table 2: Counts of data resources found by ad-hoc analysis

The only class of channels measured was port listeners. Each codebase utilizes exactly two IMAP port listeners, one on channel 143 and one on channel 993. UW does not implement its own port listener, but relies on `imapd` or another program to provide it, while Courier and Cyrus include these listeners within the codebase. All IMAP commands are sent over these channels. In addition, Cyrus contains a third port listener, defaulting to port 2000, for remote authenticated users to edit their sieve filters. Port listeners used to receive e-mail from a mailserver were not enumerated, since mail is a dependency which Courier or UW could fill in a wide variety of ways.

3.3 Enumeration of Types

A count of the set of resources belonging to each system is not sufficient for a comparison of systems because resources differ in terms of their contributions to the system's attack surface. I therefore identified a set of types relevant to each category of resources. For each resource, the types of interest are those which define (1) what restrictions are placed on the set of users who can access the resource, and (2) what privileges the resource has. [14]

For method resources, i measured:

- What subset of remote users can access the resource? Note that we are interested in the set of users who can run the code which implements the resource, not the set of users who can succeed in using the implemented feature. As an example, suppose an authenticated user tries to create a mail folder which he is not authorized to create. He will (we hope) not pass the permission check for the name of the file. That permission check, however, is contained within the code block triggered by the `CREATE` API element, so the user is authorized to run that code. However, if the codebase implements a check for whether the user is anonymous, and refuses to even run the `CREATE` code for an anonymous user, then the anonymous user would not be considered to have access to the resource.

  In general, a method can be available to: unauthenticated users, anonymous users, authenticated users, administrative users, any authenticated users (including anonymous), or any users.

- With what privilege does the code implementing the method run? This could be the root account, a dedicated imapd account, an authenticated user's account, a service account, any non-root account used by the codebase, or any account used by the codebase.

For data resources, i measured:

- File permissions, which were classified as: weak (world-writable), nonsensitive (world-readable), sensitive (not world-readable), or encrypted (some or all of file's contents are encrypted).

- Special remote access types, including whether the file type supports access control lists (ACLs) and whether the file type supports shared folders. These were only relevant for user-visible mail files.

For channel resources, i measured:

- The protocol used on the channel

- Whether SSL encryption is available and/or mandatory on the channel

- Whether the channel supports anti-denial of service protections, such as restriction by IP address and restriction of the number of simultaneous servers which may be spawned.

In addition, for all categories of resources, i measured how easy it was to disable the resource within the codebase. I listed each resource as either "cannot be disabled", "can be disabled", or "can be disabled, and is disabled by default." This allowed me to classify the three codebases in terms of their minimal configurations, as well as in terms of the full feature set offered by the codebase.

3.4 Results of Ad-hoc Measurement

This section contains the summarized results of type extension for methods and data for the three codebases. (The channel results are omitted due to the small number of data points.)

		UW	**Cyrus**	**Courier**	**TOTAL**
backend	encrypted			1	1
	sensitive	3	4	8	15
	nonsensitive	15	22	27	64
	weak				
	SUBTOTAL	18	26	36	80
user	encrypted				
	sensitive + shareable	1	1	1	3
	sensitive	3	9	5	17
	nonsensitive		1	1	2
	weak	1			1
	SUBTOTAL	5	11	7	23
TOTAL		23	37	43	103

Table 3: Classification of data resources found by ad-hoc analysis

A few items of interest are apparent from the data results in Table 3. First, the overall file counts indicate that Courier has the largest number of backend files, while Cyrus has the largest number of mail-relevant files. The latter result is not at all surprising, since Cyrus's custom mail format includes a significant amount of complex metadata. All three servers allow sharing or ACLs on mail folders. (UW supports only folder sharing, and does so in a way which requires the use of a world-writable temporary file.)

Courier's large overall filecount is primarily due to its requirement of a large number of executable files. This is because privilege separation is implemented by using different binaries to provide different functions to authenticated and to unauthenticated remote users. In addition, Courier has a relatively large number of sensitive files because most of its configuration files are only readable by root.

The methods are shown in Table 4, which contains the full codebases, and in Table 5, which contains only those methods which cannot be disabled. Unsurprisingly, most of the additional features and authentication mechanisms are optional (though some form of authentication must be used), while most of the API methods, many of which are specified by RFC, cannot be disabled.

	access	priv	UW	Cyrus	Courier	TOTAL
API method	any	any	4		3	7
	any	imapd		6		6
	unauth	root	3		3	6
	unauth	imapd		2		2
	anyauth	anynonroot	20			20
	anyauth	imapd		46		46
	auth	user	8		30	38
	SUBTOTAL		35	54	36	125
auth mech	unauth	root	5	1	4	10
	unauth	imapd		4		4
	SUBTOTAL		5	5	4	14
feature	any	root			1	1
	any	imapd		1		1
	unauth	root		1		1
	unauth	imapd		1		1
	anyauth	anynonroot	4			4
	anyauth	imapd		2		2
	auth	user	2		3	5
	SUBTOTAL		6	5	4	15
TOTAL			46	64	44	154

Table 4: Ad-hoc classification of method resources in full codebases

These results corroborate some of the findings from the observation of the software packages, most notably that UW-IMAP has a number of extra features, some of which cannot be disabled. However, one surprise is the size of the Cyrus API set — there are 1.5 times as many methods available in that codebase as in the other two, a risk which did not really stand out during manual observation.

This analysis provides no consistent comparison between the codebases. It is unclear, for example, which codebase is most attackable via its data resources: Cyrus has the largest number of metadata files, but Courier has the largest number of backend files, but UW-IMAP has files with weak permissions. The method analysis shows a similar problem, in which no codebase is most or least open in all categories, so it is unclear how to compare them.

In addition, this analysis fails to look at the code itself in any systematic way. I looked at the `main()` loop of the remote connection processing code of each codebase in order to determine what access rights were needed to run which code, but, beyond that, i did not look at which code was actually accessible to which users. Thus, while the data obtained by ad-hoc analysis is interesting, it does not allow us to rank the codebases concretely or systematically.

4 Entry/Exit Point Analysis

In this section of the paper, i 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, in

	access	priv	UW	Cyrus	Courier	TOTAL
API method	any	any	3		3	6
	any	imapd		5		5
	unauth	root	3		1	4
	unauth	imapd		1		1
	anyauth	anynonroot	20			20
	anyauth	imapd		45		45
	auth	user	8		29	37
	SUBTOTAL		34	51	33	118
auth mech	unauth	root	2		2	4
	unauth	imapd		2		2
	SUBTOTAL		2	2	2	6
feature	anyauth	anynonroot	1			1
	auth	user	1			1
	SUBTOTAL		2			2
TOTAL			38	53	35	126

Table 5: Ad-hoc classification of method resources in minimal codebases

Section 5, discuss how those results compare to those derived by ad-hoc analysis. 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 will be relevant to the results which would be uncovered by a sufficiently thorough measurement using other means.

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 production of the attack surface metric for a given codebase.

4.1 Entry/Exit Background

The entry/exit methodology endeavours 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.

4.1.1 Automating Discovery of Attack Surface Elements

Entry/exit analysis is a technique which we hope will allow researchers to automate the measurement of attack surfaces for codebases. It 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. [14]

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 either receives or transmits data from 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.

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.

4.1.2 Enabling Numerical Comparison of Attack Classes

One major difficulty with the attack classes discovered by ad-hoc analysis is that they cannot be compared to one another. However, if we use entry and exit points to find points within the code at which attack surface elements occur, our task becomes easier.

Suppose that, for each point 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, there is an obvious total ordering which exists on these elements — 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 numerical values to privilege and access levels which are consistent with the total ordering. We can then 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 $p : \{\text{privileges}\} \rightarrow \mathbb{R}$ and $ac : \{\text{access\_rights}\} \rightarrow \mathbb{R}$, and if a given item has privilege Q and access rights B, then we can define the attackability of that item as $\frac{p(Q)}{ac(B)}$. 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 consists of three very different types of elements — methods, data, and channels — and there is no straightforward way to correllate elements of these three types. Therefore, what entry/exit analysis should ideally give us is a three-dimensional vector describing the system attackability, data attackability, and channel attackability of a codebase.

4.2 Methodologies 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 also my efforts in attempting to measure data and

channel entry and exit points.

4.2.1 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` [9] will find the list of symbols which are referenced by a piece of compiled code. Then, a source code examination tool such as `ctags` [7] 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. So once the behaviour 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. Therefore, 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 calls involving data or channel access, and 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 such as "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 routines faithfully adhere to the strictest permissions possible rather than negligently substituting more relaxed permissions than needed.

Working from the other end, software observation in a jailed environment does give us access to the full list of files that each codebase requires in order to operate, but it tells us only about the files required by the very specific configuration used in that example.

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

4.2.2 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`. [8] Here is some sample `cflow` output from the UW-IMAP case:

```
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 {}
```

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 an authentication step — before that comparison, the 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. The original code fragment, from the part of the main loop which handles the `AUTHENTICATE` command, is:

```
    ...
    if (imapd_userid) {
```

```
      prot_printf(imapd_out, "%s BAD Already authenticated\r\n", tag.s);
      continue;
    }
    cmd_authenticate(tag.s, arg1.s, haveinitresp ? arg2.s : NULL);
    snmp_increment(AUTHENTICATE_COUNT, 1);
  } else if (!imapd_userid) goto nologin;
  else if (!strcmp(cmd.s, "Append")) {
    if (c != ' ') goto missingargs;
    ...
```

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

```
    ...
    if (imapd_userid) {


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


    ...
```

Note that i needed to verify that it is possible for `cmd_authenticate()` to return to the main loop upon failure. If `cmd_authenticate()` instead killed the process, or jumped to another segment of code, then the call to `snmp_increment()` would also have been unreachable by an unauthenticated user.

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.

4.3 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.

4.3.1 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 set of functions which actually make this possible?

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 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. Here is an illustrative example of my reasoning. This code fragment is taken from the UW `imap-2004a` codebase, which was reported to be vulnerable in early 2005 [11] (all code irrelevant to the attack has been excised for clarity):

```
static int md5try = 3;

char *auth_md5_server (authresponse_t responder,int argc,char *argv[])
{
  char *ret = NIL;
  ...
  u = (md5try && strcmp (hash,hmac_md5 (chal,cl,p,pl))) ? 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;
}
```

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.

4.3.2 Undercounts and Overcounts

In order to automatically enumerate reachable methods, it is necessary to have a reliable method of accurately parsing 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. 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` [6], 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` [4], 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.

4.3.3 Classifying Multiply-Accessed Methods

Once the codebase has been divided by privilege and access levels, many functions are left 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 inadvertantly 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 at 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 seemed to 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.

4.4 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.

4.4.1 Method Counts by Codebase

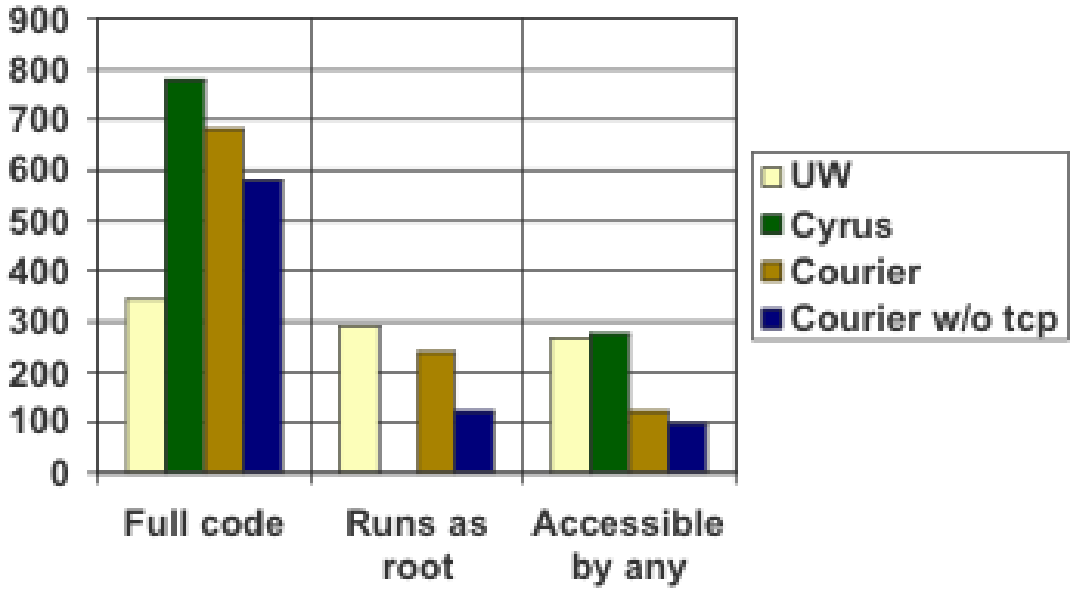

Figure 2: Number of reachable methods in each codebase: (a) full codebase, (b) code which runs as root, (c) code accessible at any access level

Figure 2 shows some 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 reachable by every access level 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.

4.4.2 Total Orderings for Privilege and Access Rights

Once all reachable code methods had been counted, 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 6: 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 7: Attackability ordering for method privileges (higher value is more valuable target)

Table 6 contains the ordering i derived for access rights, and Table 7 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.

4.4.3 System Attackability of IMAP Servers

codebase	system attackability
UW-IMAP	5044
Cyrus	5217
Courier	3122
Courier w/o tcp	1813

Table 8: System Attackability of IMAP Servers

Table 8 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 actually be less vulnerable than UW were the functionality provided to be taken into account.

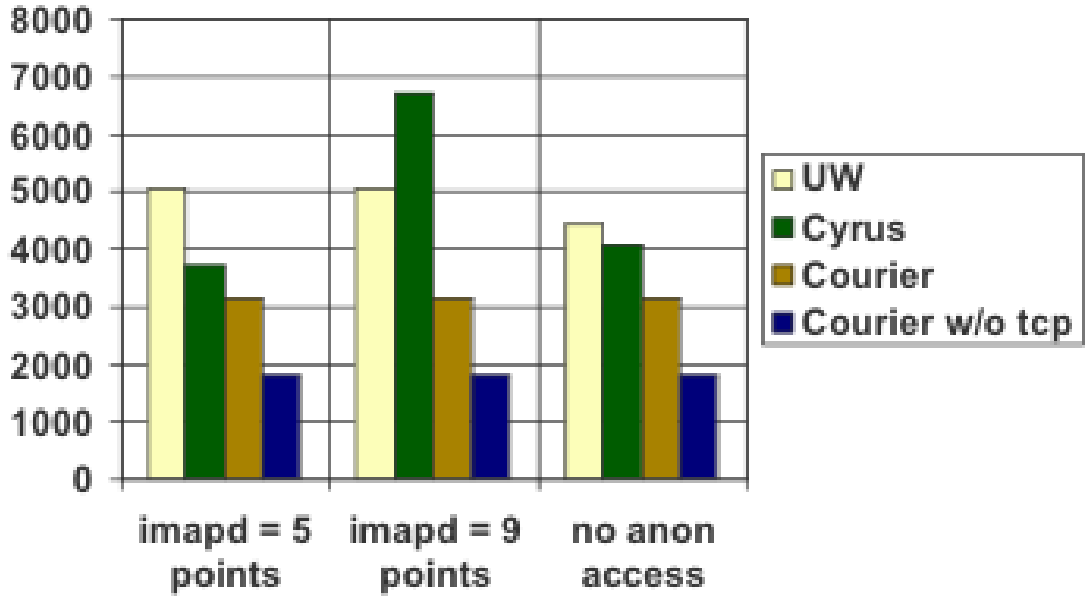

Figure 3: System Attackability of IMAP servers, with (a) imapd user worth 5 points, (b) imapd user worth 9 points, (c) imapd user worth 7 points and no anonymous access

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 7 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 5 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 9 points to imapd, Cyrus would suddenly appear radically more attackable than UW.

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 this judgment call.

In general, however, the attackability metric seems to agree reasonably well with observation. Despite the large size of the Cyrus codebase in terms of the number of methods shown in Figure 2, 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.

5 Conclusions

In order to claim that entry/exit analysis is a useful alternative to manual ad-hoc analysis for enumerating attack classes and measuring an attack surface, i need to demonstrate two things. First, i need to show that the results obtained via entry/exit analysis are at least as accurate a measurement of the attack surface as the results obtained via ad-hoc analysis. Second, i need to show that entry/exit analysis is feasible to perform in practice.

5.1 Comparison of Ad-hoc and Entry/Exit Attack Classes

The ad-hoc analysis of IMAP server methods obtained items in three categories: API functions available to the remote user, authentication mechanisms available to the remote user, and extra server features not specifically associated with any API function. The entry/exit analysis of IMAP server methods obtained items in only one category: functions reachable by the remote user. These categories are not immediately comparable, but, in both cases, the analysis broke down the elements obtained into attack classes on the bases of: the subset of remote users who could access the element (access rights), and the privilege with which the code implementing the element runs on the IMAP server machine (privilege).

Tables 9, 10, and 11 show the attack classes obtained via the two analysis types for each of the three codebases. (Note that different codebases have different sets of available users, i.e., since

UW-IMAP has an anonymous user but Courier doesn't, the access right any for UW means "unauthenticated or authenticated or anonymous", while any for Courier means just "unauthenticated or authenticated".)

Attack Class		Ad-hoc			Entry/Exit
Access	**Privilege**	**API**	**Auth**	**Feat**	**Methods**
any	any	4			269
unauth	root	3	5		21
anyauth	anynonroot	20		4	43
auth	user	8		2	11

Table 9: Ad-hoc and Entry/Exit Attack Classes for UW-IMAP

Attack Class		Ad-hoc			Entry/Exit
Access	**Privilege**	**API**	**Auth**	**Feat**	**Methods**
any	imapd	6		1	275
admin + anon + unauth	imapd				2
admin + unauth	imapd				33
unauth	root		1	1	
unauth	imapd	2	4	1	104
anyauth	imapd	46		2	325
anon + auth	imapd				1
admin	imapd				38

Table 10: Ad-hoc and Entry/Exit Attack Classes for Cyrus

Attack Class		Ad-hoc			Entry/Exit
Access	**Privilege**	**API**	**Auth**	**Feat**	**Methods**
any	root			1	10
any	any	3			112
unauth	root	3	4		102
auth	any				3
auth	root				12
auth	user	30		3	444

Table 11: Ad-hoc and Entry/Exit Attack Classes for Courier

Ideally, at each privilege/access pair, there would be a reasonable match between the combined results from ad-hoc analysis and the single result from entry/exit analysis. Largely, this not actually the case. However, both means of analysis populated approximately the same attack classes.

In the Cyrus and Courier cases, the set of API methods in each attack class corresponded somewhat well to the amount of code available to that attack class, in the sense that large amounts of code (over 100 functions) correspond to provided API methods, and more code is available to classes which provide more API methods. This is not at all true in the UW-IMAP case, in which almost all

functions are available to all users and to all privileges. This discrepancy is due to the fact that the entry/exit analysis counts which internal functions are actually available to which users, not which ones the API claims should be available. Since the UW-IMAP codebase does not have good internal privilege separation, the matchup is poor. My assessment is that, in this case, the entry/exit analysis is more accurate.

Entry/exit analysis finds more attack classes than does ad-hoc analysis, because it identifies internal code functions which happen to be available under strange subsets of privilege/access combinations (see the "admin + unauth" class in Table 10). It also identifies disconnects between change in authentication status and change in privilege, as in the Courier codebase, in which a certain amount of code is run after the user authenticates successfully, but before the `setuid()` call occurs, causing the appearance of a small authenticated/root class in addition to the expected unauthenticated/root and authenticated/user classes.

I ran the entry/exit analysis only at the subroutine level. Therefore, it is not necessarily an accurate reflection of what the code is doing on a line-by-line basis, especially for codebases which perform privilege separation poorly at the subroutine level. Entry/exit analysis might identify a function as available to both root and a user account, when the two accounts would interact with entirely different pieces of the function.

One notable omission of the entry/exit analysis is that it looked only at the binary which implemented the entry point through which the remote user communicated. This means that, in both the Courier and Cyrus cases, i did not examine the separate authentication daemon through which the server (always in the Courier case, optionally in the Cyrus case) authenticates the user. This explains a relatively major anomaly in the Cyrus case — the code which ad-hoc analysis identified as running as root was related to the authentication daemon.

5.2 Feasibility of Entry/Exit Analysis

That last point is interesting. It is entirely unsurprising that an ad-hoc analysis of attack surfaces would fail to analyse major aspects of each subsystem. However, an entry/exit analysis also suffers from the need to ensure that all components of the system have been enumerated. In this particular case, the authdaemons would have been discovered by a channel analysis, since IMAP communicates with them via socket.

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 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, even determining the total ordering among access rights might be complicated — 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 an improvement over ad-hoc analysis. It requires significant effort, but a comprehensive ad-hoc analysis requires much more research. For that effort, entry/exit analysis gives more comprehensive results which say more about the internal security posture of the codebase.

5.3 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.

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.

5.4 Comparison of IMAP Server Software

I will conclude 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 simply not designed in a security-conscious way. In comparing Cyrus and Courier, i conclude 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 introdces, Courier is likely to be the best security risk when choosing one of these three products to act as an open source IMAP server.

6 Acknowledgements

I would like to thank my advisors, 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.

References

[1] Bugtraq Mailing List. http://www.securityfocus.com/archive/1.

[2] Common Vulnerabilities and Exposures database. http://www.cve.mitre.org.

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

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

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

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

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

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

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

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

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

[12] HOWARD, M., PINCUS, J., AND WING, J. M. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security* (Aug 2003).

[13] KAMP, P.-H., AND WATSON, R. N. Jails: Confining the omnipotent root. Tech. rep., FreeBSD Project. http://docs.freebsd.org/44doc/papers/jail/jail.html.

[14] MANADHATA, P. Entry Points and Exit Points. Personal communication.

[15] MANADHATA, P., AND WING, J. M. Measuring a System's Attack Surface. CMU-CS 04-102, Carnegie Mellon University, Jan 2004.

[16] MULLET, D., AND MULLET, K. *Managing IMAP*. O'Reilly Media, Inc, 2000.

A Details of Ad-hoc Analysis of IMAP Servers

This appendix contains listings of the method and data elements enumerated during the ad-hoc analysis of the three IMAP servers. It is included to describe the specific set of features i counted in my analysis, and is most likely to be useful to readers who are familiar with the IMAP protocol and programs.

A.1 Methods Enumerated in Analysis

A.1.1 API Methods

Table 12 contains the set of supported API commands discovered for each server. These commands were found and classified by examination of each server's code to handle input from the remote user. For UW-IMAP, the routine examined was `main()` in the file `imapd.c`. For Cyrus, the routine examined was `cmdloop()` in the file `imapd.c`. For Courier, the routine examined was `do_imap_command()` in each of the files `imapd.c` and `imaplogin.c`.

	Availability	**Commands**
UW	always	`CAPABILITY LOGOUT NETSCAPE NOOP`
	unauthenticated	`AUTHENTICATE LOGIN STARTTLS`
	authenticated	`BBOARD CHECK CLOSE EXAMINE FETCH FIND IDLE LIST LSUB NAMESPACE RLIST RLSUB SCAN SEARCH SELECT SORT STATUS STORE THREAD UNSELECT`
	authenticated, not anonymous	`APPEND COPY CREATE DELETE EXPUNGE RENAME SUBSCRIBE UNSUBSCRIBE`
Cyrus	always	`CAPABILITY ID LOGIN LOGOUT NETSCAPE NOOP`
	unauthenticated	`AUTHENTICATE STARTTLS`
	authenticated	`APPEND BBOARD CHECK CLOSE COPY CREATE DELETE DELETEACL DUMP EXAMINE EXPUNGE FETCH FIND GETACL GETANNOTATION GETQUOTA GETQUOTAROOT IDLE LIST LISTRIGHTS LOCALCREATE LOCALDELETE LSUB MYRIGHTS MUPDATEPUSH NAMESPACE PARTIAL RECONSTRUCT RENAME RLIST RLSUB SEARCH SELECT SORT STORE SUBSCRIBE SETACL SETANNOTATION SETQUOTA STATUS THREAD UID UNSELECT UNSUBSCRIBE UNDUMP XFER`
Courier	always	`CAPABILITY LOGOUT NOOP`
	unauthenticated	`AUTHENTICATE LOGIN STARTTLS`
	authenticated	`ACL APPEND CHECK CLOSE COPYCREATE DELETE DELETEACL EXAMINE EXPUNGE FETCH GETACL GETQUOTA GETQUOTAROOT IDLE LIST LISTRIGHTS MYRIGHTS NAMESPACE RENAME SEARCH SELECT SETACL SETQUOTA SORT STATUS STORE SUBSCRIBE THREAD UNSUBSCRIBE`

Table 12: API methods available to remote IMAP users

A.1.2 Authentication Mechanisms

As noted in the text, all three IMAP servers provice some manner of flexible authentication mechanism, so any count of the number of ways to authenticate is necessarily incomplete, and requires some judgment calls. To the extent possible, i tried to sort authentication mechanisms so that those which ran substantially different subsets of the IMAP codebase would be counted as different methods. I enumerated the following authentication mechanisms:

UW-IMAP:

`LOGIN` using plaintext password

`AUTH=PLAIN`, defined by `auth_pla.c`

`AUTH=CRAM-MD5`, defined by `auth_md5.c`

`AUTH=GSSAPI`, defined by `auth_gss.c`

anonymous authentication

Cyrus:

`LOGIN` using a SASL password mechanism

`AUTH` using `auxprop` and `sasldb` for shared-secret auth

`AUTH` using `saslauthd` for system or PAM-style auth

`AUTH=GSSAPI` using kerberos

anonymous authentication

Courier:

`LOGIN` using an `authdaemond` plaintext password mechanism

`AUTH` using `authdaemond` and `userdb` for shared-secret auth

`AUTH` using `authdaemond` for system or PAM-style auth

`AUTH` using `authdaemond` for SQL or LDAP-based auth

A.1.3 Features Enumerated

Table 13 contains the list of special features offered by the servers which do not correspond to API methods. Note that this is necessarily an incomplete list, since it includes only features which i observed based on my analysis of the servers.

A.2 Data Elements Enumerated in Analysis

This appendix is simply a listing of all the files i enumerated in my count of the file dependencies of each server. These files were found by classifying the IMAP-related files in each of my reference `jail()` environments, and by looking through the codebases for files which are referenced if they exist.

	Description of Features Enumerated
UW	auto-detects format of user mailboxes using drivers server can act as IMAP client, allows users to access mailboxes on third-party servers contains code to handle broken IMAP clients (Netscape and Entourage) remote command execution: assumed to be a feature of the enviroment server can act as NNTP client
Cyrus	server can act as SMTP client: used by sieve contains code to handle broken IMAP clients (generic configuration flag) flexible new mail notification, provided by notifyd server proxying capability server can act as NNTP client
Courier	server can act as SMTP client: used by INBOX.Outbox feature contains code to handle broken IMAP clients (generic configuration flag) remote command execution: uses `loginexec` file server proxying capability

Table 13: Counts of extra user features found by ad-hoc analysis

Backend files:

UW-IMAP:

lockfiles:

`/var/run/inetd.pid`

`/home/`*USER*`/`*MAILBOX*`.lock`

configuration files:

`/etc/c-client.cf`

`/etc/anonymous.newsgroups`

`/etc/imapd.alert`

`/etc/imapd.nntp`

`/home/`*USER*`/.imapalert`

authentication files:

`/usr/local/openssl/certs/imapd-192.168.1.1.pem`

`/etc/passwd`

`/etc/shadow`

`/etc/cram-md5.pwd`

executable files:

`/usr/sbin/inetd` (for incoming IMAP connections)

`/usr/local/sbin/imapd`

`/usr/bin/procmail` (for incoming mail)

libraries:

`libc`

`libcrypt`

`libssl`

`libcrypto`

Cyrus:

lockfiles:

`/var/run/cyrus-master.pid`

`/var/imap/socket/imap-0.lock`

configuration files:

`/etc/imapd.conf`

`/etc/cyrus.conf`

`/usr/lib/sasl2/Cyrus.conf`

`/var/imap/msg/motd`

`/var/imap/msg/shutdown`

authentication files:

`/var/imap/192.168.1.1.pem`

/etc/passwd
/etc/shadow
/etc/sasldb2

executable files:

/usr/cyrus/bin/master
/usr/cyrus/bin/imapd
/usr/cyrus/bin/ctl_cyrusdb
/usr/cyrus/bin/lmtpd
/usr/local/sasl2/sbin/saslauthd

libraries:

libc
libcom_err
libcrypt
libcrypto
libdb-4.3
libmd
libpam
libsasl2
libssl
libwrap

Courier:

lockfiles:

/var/run/imapd.pid
/var/run/imapd.pid.lock
/var/run/imapd-ssl.pid
/var/run/imapd-ssl.pid.lock
/usr/local/var/spool/authdaemon/pid
/usr/local/var/spool/authdaemon/pid.lock
/usr/local/etc/authlib/userdb.lock

configuration files:

/usr/lib/courier-imap/etc/imapd
/usr/lib/courier-imap/etc/imapd-ssl
/usr/local/etc/authlib/authdaemonrc
/*PATH*/*TO*/shared-maildirs
/home/*USER*/Maildir/shared-folders

authentication files:

/usr/lib/courier-imap/share/imapd.pem
/etc/passwd
/etc/shadow
/usr/local/etc/authlib/userdb.dat
/usr/local/etc/authlib/userdbshadow.dat

executable files:

/usr/local/sbin/courierlogger
/usr/local/sbin/authtest
/usr/local/libexec/courier-authlib/authdaemond
/usr/local/bin/courierauthconfig
/usr/lib/courier-imap/bin/imapd
/usr/lib/courier-imap/bin/couriertls
/usr/lib/courier-imap/bin/deliverquota
/usr/lib/courier-imap/libexec/couriertcpd
/usr/lib/courier-imap/sbin/imaplogin
/usr/bin/procmail (for incoming mail)
/home/*USER*/Maildir/loginexec

libraries:

libc
libcourierauth
libcourierauthcommon
libcourierauthsasl
libcrypt

libcrypto
libltdl
libssl

User files:

UW-IMAP:

user mail files:

/var/mail/*USER*
/home/*USER*/*MAILFILE*

mail metadata files:

/home/*USER*/*MAILBOX*/.mxindex (used by MX driver)

temporary files:

/tmp/*RANDOM*
/var/tmp/tmp.*RANDOM*

Cyrus:

user mail files:

/var/spool/imap/user/*USER*/*MAILBOX*/*N*.

mail metadata files:

/var/imap/db/__db.00*N*
/var/imap/db/skipstamp
/var/imap/annotations.db
/var/imap/mailboxes.db
/var/imap/deliver.db
/var/imap/user/*USER*.seen
/var/spool/imap/user/*USER*/*MAILBOX*/cyrus.header
/var/spool/imap/user/*USER*/*MAILBOX*/cyrus.index
/var/spool/imap/user/*USER*/*MAILBOX*/cyrus.cache

temporary files:

/var/tmp/*RANDOM*

Courier:

user mail files:

/home/*USER*/Maildir/.*MAILBOX*/cur/*FILENAME*

mail metadata files:

/home/*USER*/Maildir/.*MAILBOX*/maildirfolder
/home/*USER*/Maildir/.*MAILBOX*/courierimapacl
/home/*USER*/Maildir/.*MAILBOX*/courierimapuiddb
/home/*USER*/Maildir/courierimapuiddb
/home/*USER*/Maildir/courierimapkeywords/:list

temporary files:

/tmp/courier.lock