CS 486 – Capstone Project

Final Project Report

(Revision 1.0)

Submitted to

Dr. Doerry

By

Team Fugu:

Erik Wilson

Ben Atkin

Nauman Qureshi

Thad Boyd

On

April 30, 2004

Table of Contents

1. Introduction

On the 5th of January 2004 the United States Geological Survey (USGS) approached Northern Arizona University's (NAU) Design4Practice program with the goal of developing two time-saving tools for OpenBSD. The proposed tools would automate two time-consuming processes that previously required manual interaction under OpenBSD. On the 16th of January 2004, Team FUGU was formed of four capstone students who combined to develop this project.

A Requirements document was accepted by our client, Ernest Bowman-Cisneros and Margeret Johnson of USGS, on the 18th of February. Following that, we produced a Functional Specification document that was accepted on the 10th of March. Based on the functional specifications, we produced a design document that we delivered on March 15th.

Since the delivery of that document we have been hard at work implementing and testing our automated installation and patching products. We have completed this phase, and thus concludes our project. We have created this final report with two main purposes in mind. The first is to communicate our successes and shortcomings at the end of this project. We do this by providing an overview of our project requirements, and specific testing results to show to what level we met these requirements. The second purpose of this document is to show how we arrived at our destination. To this end, we provide an overview of how we understand the problem, and the progress of all phases leading up to the completion of our project. This ties into the last part of the report, where we document how our work throughout the semester contributed to the quality and completeness of our final solution.

2. Problem Statement

Our sponsors for this project are Ernest Bowman-Cisneros and Margaret Johnson, system administrators at the Flagstaff Field Station of the United States Geological Survey. The Flagstaff Field Station serves as the headquarters for the Astrogeology Department of USGS. The people working there are primarily involved in planet mapping and space exploration. Their research efforts in these areas place heavy demands on computing infrastructure. To manage a large number of servers, Ernest and Margaret have learned to work efficiently and make use of the latest and greatest system administration tools.

These tools include automated OS installers and automated patchers. An automated installer provides the administrator the opportunity to save time over a conventional installer, by allowing them to specify the options in a file, rather than type them in at each terminal. When using a conventional installer, the user answers questions, waits for part of the installation to finish, and answers more questions. This process makes it hard for them to get any other work done. With an automated installer, they can spend a few minutes working on the configuration file, boot a computer, leave, and return to the computer later to find the installation completed. This makes it easy to install multiple computer systems simultaneously.

Patching is a method used to plug security holes when they are discovered. When there is a known security hole in the operating system, the computer is at risk. To prevent the system

from being compromised, security patches should be installed as soon as possible. Without an automated patcher, this must be done manually. People often forget to install patches, and before they get a chance to install a patch their system is compromised. An automated patcher, however, can be instructed to check every day for a security update. The benefit of an automated patcher is two-fold: it is more convenient and it provides for better security.

Automated patchers and installers have long been available on Red Hat Linux, SuSE Linux, and Solaris – three platforms which our client, USGS, often uses. The tools shipping with these systems work well for our client. The problem is that they would like to increase their usage of OpenBSD, a free UNIX variant they have become quite fond of, but OpenBSD lacks these time-saving tools. This is where we stepped in.

They presented us a request for a proposal (RFP), and we researched the problem and submitted a proposal that matched what they were looking for. We took their recommendation to start with what was already available, and expand it into a product that matched their needs. We started with two open-source tools: the standard OpenBSD installer and Tepatche, an automated patcher that runs as a regularly scheduled task that finds, downloads, and compiles patches into the system.

We would expand the OpenBSD installer to be able to read configuration files, make some basic disk calculations, and run pre- and post-install scripts. We would expand Tepatche to create and download binary patches, a capability it did not have before that was requested in the RFP. Binary patching allows the code to be compiled only once, and then distributed to other machines, which streamlines the process.

With our creation of this product according to the principles outlined in the RFP, USGS will now have the advantages of OpenBSD – security, openness, and simplicity – as well as the convenience that until now only came with the more established systems. With these tools, rather than spend an hour patching and an hour installing on twenty machines (2 hours x 20 = 40 hours), they can spend two hours preparing a configuration file and booting the machines. The rest of the task will be completed automatically. The value of our solution is in the ability to save hours of time, and the benefit of increased security through regular patching.

3. Process Overview

3.1. Team Organization

Each person on our team was given two special responsibilities. The members and their roles are as follows:

1. **Erik Wilson - Leader and Organizer**
   Erik is the leader by default. His final decisions can only be overruled if all others vote against him (simple majority). He also arranges additional meeting times.

2. **Ben Atkin - Communicator and Researcher**
   Ben was given the task of communicating with our client. He also researches things that

are important to our project (such as how to use CVS).

3. **Nauman Qureshi - Recorder and Documenter**
   Nauman was given the task of sending out minutes after each of our meetings. He also kept our project notebook.

4. **Thad Boyd - Facilitator and Website Maintainer**
   Thad's job was to facilitate any arguments that would arise. He also designed and updated our team website.

3.2. Project Management

At the start of our project, we agreed on some by-laws, that provided structure for our team, that helped us through our project. We agreed to meetings three times a week. Nauman agreed to record significant decisions in the meetings, in an e-mail sent to everyone else in the team. We also created a Gannt Chart to keep track of deadlines. These made up most of our reporting and monitoring processes. We also had the project notebook, but most of the time it did not make it to our meetings. Our website and e-mail partly made up for it.

To keep the project going smoothly, we also agreed on standards of interoperability. We agreed to send our e-mails in plain text, and to make our documents available in three different formats. These rules, when properly practiced, simplified the sharing of information between each other and our client.

We decided on the simple majority rule for making decisions, but it was never used. Occasionally we got into debates about the design, and when someone suggested putting it to a vote, the arguing parties agreed to make a compromise.

Regular meetings and minutes helped offset our lack of memos and use of our notebook. If we had implemented these better, we could have saved a lot of time in the long run.

3.3. Design Methodology

We did not spend a lot of time on the design methodology. Erik wanted to try out SCRUMS, which is an agile development method, that is focused on removing obstacles and having better communication, especially with the client. SCRUMS involve “sprints”, where a team works for 30 day periods, to get part of a project done. There is daily reporting and a person representing the client is on-site.

We chose to use SCRUMS, but in reality we followed more of a waterfall method. We didn't do any major coding until the design document was done. We did follow one aspect of SCRUMS – regular meetings. At our meetings we discussed our progress, and were able to keep our project on track. We did have some slippage, mainly because we often focused on short-term goals (such as specific deliverables) rather than long-term functionality goals. This resulted in us doing less testing than we would have liked.

This was the weak point in our project organization. Perhaps we would have done well to select the waterfall or spiral design methodologies for this part-time working environment.

3.4. Deliverables

Deliverables were used to track our project through completion. We had deliverables that guided our project from drawing board to implementation. These followed the pattern below:

Project Requirements → Functional Specifications → Detailed Design → Delivered Product

Each of the deliverables in this category directly built on the previous concept. With each deliverable, we obtained a better understanding of the final solution we are providing. We started with the customer concept, by listening to our client give us their vision for the product. After we understood that, we obtained specific requirements. When creating the Functional Specification, we determined how we would like our product to work, to best satisfy the requirements. We moved into the Detailed Design, which we used as a guide to create our Delivered Product.

The remaining deliverables were designed to streamline our project management and development processes. These deliverables included our Team Inventory, Team Standards Document, Project Development Plan, and our Usability and Functional Testing Plans. As we created the first three documents in this list, we adopted good practices that helped us in the completion of our project. One of these was version control. We found it very useful in collaborating on our document, and in not worrying about saving previous versions. The last deliverable (besides this one) was our Usability and Functional Testing Plans. We used that in our development process to ensure quality.

We found that each deliverable contributed to our success. We also found that some of our difficulties could have been adverted had we better applied our documents to our design and implementation process. The following table provides a basic description our deliverables, and how they contributed to our success. They also contain brief notes on our shortcomings with regards to putting the deliverables into practice.

Deliverable	***Purpose***	***Successes***	***Shortcomings***
1. Team Inventory *2004-1-26*	Document the skills we have available to us, as a team, that are pertinent to this project.	Provided a good introduction of us to our client.	
2. Team Standards Document *2004-1-28*	Contains rules that will help us govern our project.	Established regular meeting schedule. Used CVS as planned; worked very well for us.	Poor intra-team documentation. Should have been more specific. Did not keep up with self-evaluation.

Deliverable	*Purpose*	*Successes*	*Shortcomings*
3. Project Devlopment Plan *2004-2-2* *2004-2-16 (revised)*	Decide how to gather requirements and analyze risks. Decide which hardware and software tools to use.	Thought about testing equipment, and was able to plan well there. Set up OpenBSD on a number of different computers. Provided excellent framework for future documents.	
4. Project Requirements *2004-2-18*	Get the project requirements from the client, and begin to lay out a plan to achieve them.	Became clear on what was expected of us.	Did not keep up on risk management after document was complete.
5. Functional Specifications *2004-3-1*	Determine, more specifically, how the program is expected to function, including user interface details.	Better understood workings of an installer and a patcher.	Did not reference often enough.
6. Software Design Specification *2004-3-15*	Create an internal document with design specifics, that will guide us while we implement our program.	Understood installation and patching, step-by-step.	Was not fully in-line with some of our goals, such as being able to interrupt the install. Few things were kept in the implementation.
7. Usability & Functional Testing Plan *2004-4-12*	Determine what will be a part of usability and functional testing, and lay out plan to implement it.	Made some needed usability improvements beforehand.	Slow to begin usability tests. Had little time to make improvements based on usability testing.
8. Final Capstone Report *2004-4-30*	Document and analyze the outcomes of our project.	Better understanding of engineering process.	???

This table shows a pattern: when we followed the documents, we experienced success. In retrospect, we decided that it would be a good idea to break out our completed documents in our meetings. This would have helped us to see eye to eye.

3.5. Timeline

Our project began with creating our team documents and discussing what the project entails and how best to complete the project. We discussed this with our client on several occasions in January. By mid February, we had nearly all of our requirements obtained from our sponsor and had described them in the Requirements document. With half a month's time to finalize the design that we intended to use, we were at the end of our first super sprint. By this time we had started coding, this was in the first week of March. Along with minor coding, we worked more on our design in order to make it as efficient as

possible as we were getting closer to our deadline. With the end of our second super sprint in the first week of April, we had an almost fully functional product ready to be tested though with some minor bugs needed to be fixed in it. On the 5th of April we had started to test our product by drafting up scenarios and playing them out. After we were satisfied with the basic functionality, we conducted usability testing.

Our timeline of significant events is shown below. Our completion of major functionality, and our testing each slipped back a full week. Our design document was also a weekend late. Other than that, we made all of our deadlines. Following the table is an explanation of why the slippage occurred.

No.	***Date***	***Event***
1	*2004-1-21*	First meeting with client at USGS Flagstaff Field Station
2	*2004-1-26*	Completed Team Inventory deliverable
3	*2004-1-28*	Completed Team Standards Document
4	*2004-2-2*	Completed Project Development Plan
5	*2004-2-16*	Revised Project Development Plan
6	*2004-2-18*	Completed Project Requirements deliverable
7	*2004-3-1*	Completed Functional Specifications deliverable
8	*2004-3-1*	Coding Begins
9	*2004-3-15*	Completed Software Design Specification
10	*2004-4-1*	Most of the Functionality is Complete
11	*2004-4-9*	Began Testing for our Products
12	*2004-4-12*	Created Usability and Functional Testing Plan
13	*2004-4-21*	Submitted working installer and patcher to client for testing
14	*2004-4-23*	Capstone Presentation
15	*2004-4-30*	Completed Final Report

Partly because of our lack of structure in our design methodology, and partly because of the learning curve that comes with learning a new operating system and two new programming languages, we did not get any major prototypes working until after the design document was complete. This made it a strain to get our product complete. We worked hard at it, however, and got all of the basic requirements complete. From the full requirements and functional specifications, there are a couple of things that would have been useful that weren't completed. In retrospect, it would have been better to have put off some parts until later, and tested our code in smaller modules. If we did, it might not have taken a couple of days work to satisfy a simple requirement.

Our client, however, is happy with our product. There will no doubt be some bugs to work out, but it is useful in its present state. We have declared our project to be a success.

4. Project Requirements

4.1. Overview

We determined the requirements for our project through frequent team meetings and frequent communication with the sponsor for refinement and clarification. We communicated with the sponsor primarily through E-Mail, but met face-to-face on several occasions as well.

The primary building block for the project requirements was, appropriately enough, our requirements document, available in our team notebook and on our team website. We submitted a draft to our sponsors, and then met with them to discuss changes and refinements to the requirements we had listed.

4.2. Project Goals

The keyword for both projects' goals is *automation*. We set out to make both projects as convenient, time-saving, and hands-free as possible. We designed the automated installer to duplicate the functionality of the existing manual OpenBSD installer but added the capability of taking input from a configuration file rather than requiring keyboard input throughout the installation process. For the automated patcher, we updated Tepatche to handle binary packages. We designed both programs to run on i386 or Sparc architecture, and believe that they could easily run on other platforms as well.

4.3. Summary of Functional Requirements

What follows is a condensed summary of the major functional requirements we determined. A more detailed version is available in our requirements document.

4.3.1. Automated Installer Requirements

1. Must be substituted into existing OpenBSD installation environment.
2. Must be able to handle either clean install or upgrade and existing OS installation.
3. Must read an installation configuration from multiple sources.
4. Must be able to handle all the installation steps currently performed by the manual installation process.
5. The tool must also handle pre- and post-installation scripts.

4.3.2. Automated Patcher Requirements

1. It must handle both source and binary patches.
2. Binary patches must be able to run a pre-install, post-install, pre-uninstall and post-uninstall scripts, and contain install and uninstall processes.
3. The patch system must keep track of what patches have been installed.

4. The system must be able to send an email, using the standard UNIX mail system, to the system administrator(s).
5. A highly desirable feature is that the tool be able to run in the installation environment.

4.4. Summary of Performance Requirements

What follows is a condensed summary of the major functional requirements we determined. A more detailed version is available in our requirements document.

1. User interaction should involve as minimal amount of time as possible.
2. There is no requirement for interactive processes to configure the automated installation or patching systems.
3. There is no requirement for the amount of time the automated installation process must occur in.
4. There is no requirement for the amount of time the automated patcher must apply the patches.

4.5. Constraints

One of the main advantages of OpenBSD is that it is easily installed and can run on legacy systems. The ramdisk installation method requires a minimal amount of memory to load the ramdisk (approximately 8 megabytes), however there are other installation methods to circumvent this requirement. OpenBSD can run on Alphas, HP300 (and above), Intel's i386 (and above), and many other such architectures. This takes care of most of the hardware constraints. There are no other forms of constraints applicable other than the ones mentioned above.

5. Solution Statement

In this section we outline our completed solution. We begin with an overview of our solution. Following that we document how we arrived at our solution with a functional specifications summary and a summary of the design in our design document. We conclude with an as-built statement.

5.1. Overall Solution

Our solution was to modify two existing products, the OpenBSD installer and Tepatche, to meet our sponsor's needs.

5.1.1.Automated Installer

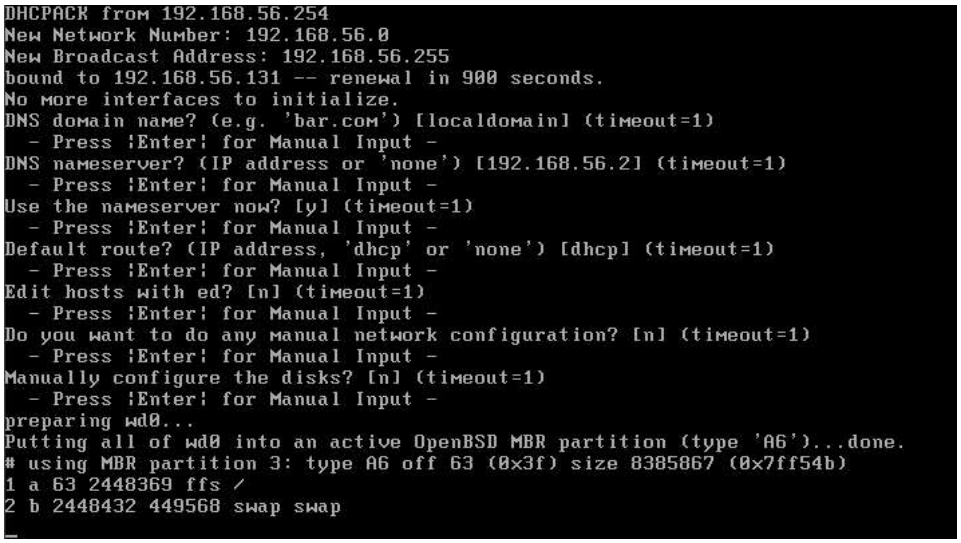
Modified from the original shell scripts of the existing OpenBSD installer, our automated version allows for the same functionality of a series of prompts and adds a number of

features. Most importantly, our automated installer will answer the prompts based on settings provided in a separate configuration file. However, thinking in terms of integration with the OpenBSD project as a whole, we went above and beyond this sponsor requirement and added an optional interactivity feature: at every prompt, a timeout in seconds may be specified in the configuration file, and until that timeout elapses, the user may interrupt the interactive install and choose a custom option.

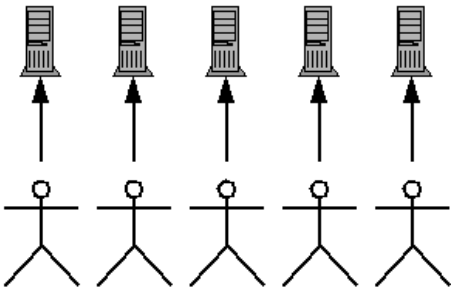
This is best demonstrated in the screenshot below (5.1), which shows a series of prompts with default options specified in brackets and timeouts specified in parenthesis. For example, the line that reads,

```
DNS domain name? (e.g. 'bar.com') [localdomain] (timeout=1)
```

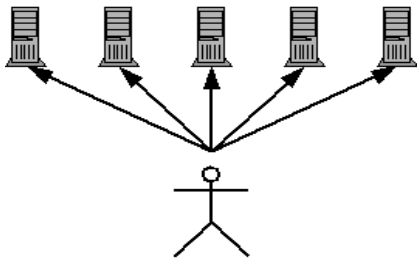
will choose the default, "localdomain", unless the user presses the Enter key before the 1-second timeout elapses and enters a custom option.



5.1. The automated installer in action.



5.2. Manual Installation and Maintenance



5.3. Automatic Installation and Maintenance

The automated installer solves the problem of time-consuming system installation. By generating a single configuration file for installing multiple machines simultaneously (figure 5.1), the system administrator is able to specify installation settings in a fraction of the time it would take with a fully manual installation (in figure 5.2).

5.1.2.Automated Patcher

Our automated patcher is modified from the original Tepatche Perl scripts coded by Gunnar Wolf. The key features we kept from Mr. Wolf's version are the ability to read settings from a configuration file (tepatche.conf) and the ability to download, apply, and compile source patches. The major functionality we added was the ability to package these compiled binaries in OpenBSD's .tgz package format and choose to install such binary packages rather than source patches.

The patcher solves the problem of time-consuming system maintenance, and solves the problem of repeated compilation inherent in the original Tepatche. Run as a cron job, our patcher will keep the system updated without requiring user input, and, with our extensions, can install existing binary patches rather than being limited to the wasteful process of patching and compiling each program from source.

5.2. Functional Specifications

Here we provide a quick overview of the functional specifications for our product. These are listed in groups, by the tool they pertain to.

1.1.1. Automated Installer

1. The automated installer shall be able to do a fully automated installation or upgrade of a system.
2. The automated installer shall work on i386, Sparc, and Sparc64.
3. The automated installer shall be able to download configuration files from multiple locations, including ftp.
4. The automated installer shall be capable of everything that the standard OpenBSD installer is capable of.
5. The automated installer shall provide the option to run pre- and post-install scripts.

1.1.2. Automated Patcher

1. The automated patcher shall be able to create binary patches from source downloaded off OpenBSD mirrors.
2. The automated patcher shall be able to download binary patches from an FTP server.
3. The automated patcher shall be run as a regularly scheduled task.

4. The automated patcher shall be able to handle source patching.
5. The automated patcher shall get its options from a configuration file.

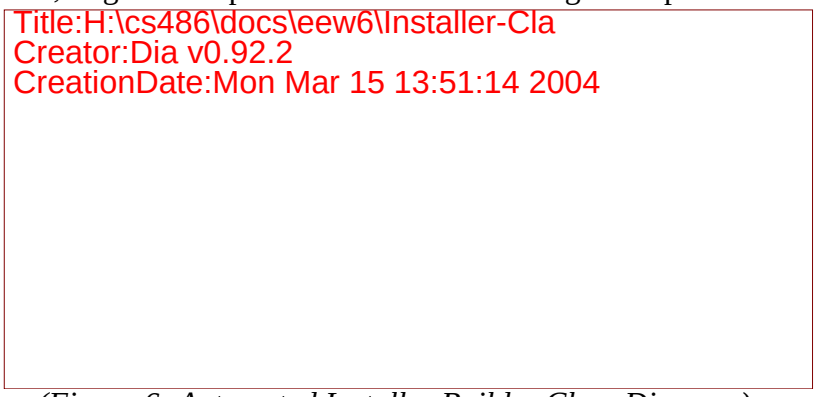
These are taken from the Functional Specifications document. We communicated with our sponsor to obtain these functional specifications, and they shall serve as a measure of the success of our project. The above functional specifications are covered in the functional testing section of this document.

5.3. Architecture Overview

This section provides a brief summary of the architecture we proposed in the Software Design Specification.

1.1.3. Automated Installer

The Installer Runtime installs an OpenBSD system using information provided in a configuration file, as well as decision trees carefully researched, implemented, and tested by our team, to guide the process. The modular design is depicted in the image below:



(Figure 6: Automated Installer Builder Class Diagram)

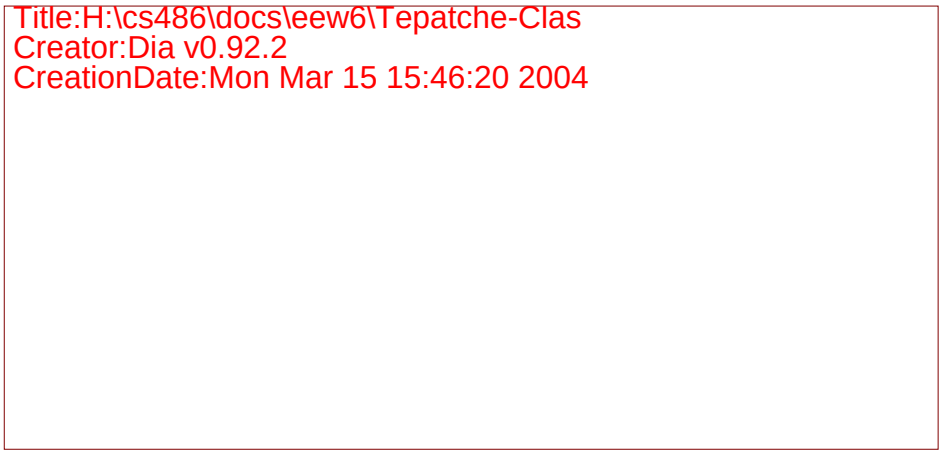
As depicted above the *dot.profile* script is used to determine whether to launch the *install.sh*, *upgrade.sh*, or *auto_inst.awk* script. The automated process will use the *config_parser.awk* script to access the values from the configuration files which it finds.

The Installer Runtime has a large number of dependencies. It depends on the kernel to properly recognize devices. It depends on the networking programs to help it to determine the proper network interface and get the network up. It also depends upon the partitioning and slicing tools. The building and adding of these things to the disk images will be taken care of by the disk image build scripts. For the Installer Runtime to work, it is important that the Disk Image Builder compile all the tools and place them in the right locations.

1.1.4. Automated Patcher

The following Figure 12 depicts the overall interactions within Tepatche:

(Figure 12: Tepatche Class Diagram)



As depicted above Tepatche is divided into three distinct parts: a Binary module, a Source modules, and the Main module. Several important functions are depicted in the previous classes:

- The ability to download source tarballs if they are needed.
- The ability to download source or binary patches as needed.
- The application to the system of those patches.
- The ability to create a binary patch given that the source is patched.

Tepatche will be installed during the post-install phase listed in Section 5.2.6 and will be run as root during a periodic cron job. A configuration file will also be used for Tepatche in which the script will determine which patching type it will perform (binary or source), if it should host binary patches, as well as various other configuration options such as logging level and email address to send messages to.

5.4. As-Built Design

5.4.1. Automated Installer

5.4.1.1. Overview

We built the automated installer from the source code of the original interactive installer, licensed under the BSD license. We designed our code to be run in basic shell script, using the minimal ramdisk environment provided by the original installer. All of the programs available in this environment can be contained on a floppy. Programs such as perl and awk were not available; we had to use a mix of shell and sed.

After looking in detail at the source code of the original installer, we saw that the logic for getting input from the user was simple and uniform. Two subroutines beginning with “ask” handled most input situations. One took a single value from the user, optionally providing a default value, which could be selected by pressing the enter key. The other subroutine allowed the user to select from a list of values.

After thinking about it for a while, we decided to co-opt the reading of a configuration file into the existing interactive installer script. We modified the “ask” routines to wait for a time specified in the configuration file, allowing the user to press enter and type the in value manually, before timing out and going with the “default”. The “default” in this case is taken from the configuration file, using a section and key name, as the third and fourth parameters in the “ask” routines. If that value is not specified in the configuration file, it takes the second parameter in the ask routine to be the default.

This worked for many simple options. We chose sensible section and key names for these options. To do the actual reading from the configuration file, we wrote a sed script. We wrote another sed script that converts a string into a case-insensitive regular expression, allowing section and key names to be case-insensitive. All of these features were provided in our “util.sub” module.

We created another module, “disks.sub”, to handle the complex task of properly allocating disk space, based on values specified in the configuration file. This works by reading options from the configuration file to determine which OpenBSD “slices” (disk partitions, such as partitions for /usr or /home) go on each disk. To determine what disks are available, it uses the boot messages. Once the locations of slices are determined, the sizes are determined. This is done by reading the size options from the configuration file, and disk geometry from the disklabel command, and performing some calculations on them. The calculations are written in shell, which only has integer support, so there is an integer scaling function. After the geometry is fully decided, a disklabel, specifying the partitioning information, is added for each disk. When that is loaded, formatting of filesystems is done, device driver instances are created, mounting information is stored, and disks are mounted.

The installer (install.sh, upgrade.sh, install.sub) has loops for input, so we found ways to deal with those. These loops are present in the networking and filesets (programs to be installed) sections. We created loops of our own, that go through all values in the configuration file.

We also added pre- and post-installation script capabilities, as well as the ability to run our patcher from within the automated installer. We were able to satisfy all of the basic requirements.

We added a little script that was not a part of our original design – a CGI script to display the results of an installation on the Internet.

What we ended up with is a complete automated installer, that can work from boot to reboot, and can handle a variety of different configurations. Our sponsor tested it about two weeks ago, and they were pleased.

5.4.1.2. Modules

The modules included in our final design are as follows:

1. **util.sub**
   Contains utility subroutines, such as retrieving the configuration file, reading the

configuration file, and logging the installation.

2. **disks.sub**
   Contains code to determine the configuration and geometry for the OpenBSD slices that will be created on the fixed disks. A non-interactive process; it cannot be interrupted, either the disk setup must be fully automated or fully interactive. This is because the interaction is handled by disklabel, which is difficult to script with our minimal environment.
3. **install.sh**
   A long script, executing commands in an order that will leave the user with an automated install. For many parts, we simply added a section and a key to allow the installer to read from a configuration file. For some of the more involved parts, we created a custom loop.
4. **upgrade.sh**
   Similar to install.sh, but upgrades a system without effecting user documents. Uses “ask” routines to read from a configuration file.
5. **install.sub**
   Contains subroutines, and subscripts common to the installer and the upgrader. These include subroutines for loading the network and installing filesets.
6. **install.md**
   Modified to add room for the Master Boot Record on the i386 platform, while utilizing the full disk on the Sparc64 platform.

5.4.1.3. Changes from Original Design

Because we decided to simply co-opt the reading of the configuration file, we lost some of the structure behind our old design. We moved from having five of our own modules to having the ones shipping with the interactive installer plus two of our own.

One thing that did not get built was the configuration file validator. We did, however, design our code to accept reasonable defaults if something could not be found. None of the human errors we encountered during usability testing would be prevented by using a validator.

We also decided to stick with the original Makefiles for building disk images, as they work well for most tasks.

For the most part, our design was on par with the design document. We created special routines for setting up of disks, and for everything else, orchestrated the loading of values from the configuration file and the running of programs with the correct parameters to complete an installation automatically.

5.4.2. Automated Patcher

6. Usability Testing and Future Work

6.1. Usability Testing

Our usability testing strategy is three-fold. We have functional tests, usability tests, and acceptance tests. The three types of tests are closely related. The functional test is a test we complete ourselves, to determine which required functionality is working. The usability tests are done as an extension of the functional test, to see if the basic functionality can be completed by someone who is new to our product, with minimal documentation. Finally, the acceptance tests are tests that our client takes, to know how satisfied they are with our product.

6.1.1. Functional Tests

When we first completed a successful automated installation, the functional testing period for our automated installer began. This was on April 9. Coincendentally, our first successful binary patch was on April 9 as well, and so began the functional testing for our automated patcher. Following is a chart with specifications taken from functional specifications section within our solution statement. The chart displays who administrated the latest test for each specification, and what the outcome was.

No.	*Product*	*Test*	*Conducted By*	*Outcome*
1	*Automated Installer*	The automated installer shall be able to do a fully automated installation or upgrade of a system.	Ben	Successful
2	*Automated Installer*	The automated installer shall work on i386, Sparc, and Sparc64.	Ben	Successful
3	*Automated Installer*	The automated installer shall be able to download configuration files from multiple locations, including ftp.	Erik	Successful
4	*Automated Installer*	The automated installer shall be capable of everything that the standard OpenBSD installer is capable of.	Ben	Successful
5	*Automated Installer*	The automated installer shall provide the option to run pre- and post-install scripts.	Ben	Successful
6	*Automated Patcher*	The automated patcher shall be able to create binary patches from source downloaded off OpenBSD mirrors.	Erik	Successful

No.	*Product*	*Test*	*Conducted By*	*Outcome*
7	*Automated Patcher*	The automated patcher shall be able to download binary patches from an FTP server.	Erik	Successful
8	*Automated Patcher*	The automated patcher shall be run as a regularly scheduled task.	Erik	Successful
9	*Automated Patcher*	The automated patcher shall be able to handle source patching.	Erik	Successful
10	*Automated Patcher*	The automated patcher shall get its options from a configuration file.	Erik	Successful
11	*2004-4-9*	Began Testing for our Products		Successful
12	*2004-4-12*	Created Usability and Functional Testing Plan		Successful
13	*2004-4-21*	Submitted working installer and patcher to client for testing		Successful
14	*2004-4-23*	Capstone Presentation		Successful
15	*2004-4-30*	Completed Final Report		Successful

As we completed this test, we brought our product closer to being usable. These were an important part of the development process.

6.1.2. Expert Review

The software that we developed will require more than just basic knowledge of computers to operate it. Because installing and maintaining UNIX style operating systems are more in-depth than other operating systems, we require that our expert reviewer be more trained in the system administration aspects in order to correctly operate this software. For this reason Tom Baca, the IT manager for the College of Engineering was chosen for the expert review. The process occurred by taking notes on his opinion of the configuration files, utilities, and run of the installation and patching. The following is a list of the major suggestions which he made:

1. The logging website should have indicators on the main site for each computer if there have been any errors, if the installation is in progress, or if it has been completed.
2. The logging website should list the log entries in reverse chronological order.
3. The logging website should combine the question and the answer into a single entry on the log.
4. The logging website should have separators indicating different installations.
5. The logging website should provide a summary of the status for the last installation at the top of the page.
6. The configuration file for the automated installation should have greater detail on

the disk partitioning section (i.e. better comments).

7. When the installation completes the automatic rebooting of a system should be optional because some systems will restart the installation process, depending on their BIOS capabilities.

All of the suggestions for the configuration file and installation systems were taken into account and enacted. Our reviewer thought that the most useful tool was the installation logging system, which he offered many suggestions for improvement to the user interface. Ultimately the changes for the installation logging system never occurred due to time limitations and that the logging system was a tertiary part of the project. Any changes to the installation logging system have been left to the next revision of the

6.1.3. Usability Tests

The main objective of testing our product before handing it over to the client is to ensure that our product is fully functional and bug-free. To avoid the mishap of an unusable product we had created three testing groups who would be conducting the assessment studies for the program. The following were the groups which participated in the studies:

Group 1) Brian Adams, Shanadeen Begay

Group 2) Daniel Headly, Scott Hancock

Group 3) Jay Anderson, Erica Liszewski

The intention of these groups were to engage in constructive interaction without the aid of any designer or facilitator intervention. Each group was allotted approximately 15 minutes to perform their tasks. To best simulate realistic domain tasks the following items consisted of our lab manual:

Scenario

You are a system administrator for the United States Forest Service in Flagstaff, AZ. At one of their stations research is being done about the effect of bark beetles on our ecosystem. To help with computing power, they need six OpenBSD systems to be installed. You will be using the OpenBSD automated installer to complete this task.

Tools Available

A virtual computer is made available for testing using VMware. You will need to push play to boot it up. The physical floppy disk will be used on the virtual system, but

when you boot it up, make sure all files and directories on the floppy are closed. Otherwise, the virtual machine will not be able to secure full control of the floppy drive.

In VMWare, when you start the virtual machine, the output will be shown in a virtual console in the window. To type into the virtual console, click inside the screen. To type elsewhere, press CTRL and ALT at the same time, and release.

To edit the file on the disk, use NoteTab. Windows and UNIX files are incompatible, because Windows files use an extra character to separate lines. NoteTab, unlike notepad, which ships with Windows, is equipped to handle this.

There is a log application at http://www.cet.nau.edu/~fugu/log/logger.pl. Read the log here.

List of Instructions

1)Modify the hostname to be 'rex' in the network section.

2)Modify the configuration file under the “Disks” section to partition a disk with a 500 megabyte swap, 1 gigabyte root partition, and the remaining space allocated to the home partition.

3)Under the “Tepatche” section define the appropriate key to use your email address as the location to send logging messages for the patching process.

4)Close the editor for the configuration file and any other programs that make use of the floppy disk. If any are open, the virtual machine will not be able to use the floppy, to read the configuration file.

5)Start the virtual machine, by clicking the play button in the VMWare.

6)From the installation logging website (http://www.cet.nau.edu/~fugu/log/logger.pl) monitor the progress of the install.

7)When the installation has been completed please record the number of errors and the severity of the errors from the installation logging website.

8)From the email address defined in step 3 check our email to view the patching log. Record how many and the types of errors which occurred.

The results from the tests were interesting, these results are listed for each pertinent step as follows:

Step 1)

All teams were able to successfully complete the first step of modifying the hostname without any issues, this suggests that the configuration file format was

trivial to navigate.

Step 2)

The first team was presented with a disk configuration which was almost identical to the desired configuration with the exception of different geometries (disk sizes). They were able to successfully make the requested changes to the configuration file.

To complicate the process the next two teams were given configurations containing an extra partition. The teams did not remove the extra partitions but the second team did correctly configure the disk geometry. This suggests that further instructions needed to be given in the instructions or the configuration file, or that the groups lacked the system administration skills needed to identify the change that needed to be made.

The last team was given a further modified configuration file with the extra space going to a partition other than the desired one. They were able to successfully modify the geometry for the swap partition, however, the extra partition was not removed, the size of the root partition was not changed, and the remaining space was not correctly allocated to the home partition. These issues are however most likely due to their limited Unix administration experience and were not taken into consideration for revising.

Step 3)

All teams were able to successfully complete step three. This reinforces our findings from step 1, that the configuration file was easy to navigate.

Step 6)

The teams were able to successfully navigate the installation site. All of the teams attempted to view the installation logs before the network for the installation was brought up. This created some confusion for the teams attempting to monitor the progress of the installation. The teams often had to reload the logs manually as there was no mechanism for automatic refreshing (this will be left to the next revision of the software). The teams mostly monitored the progress of the installation as it occurred in front of them. Further information about the installation logging is provided in the next section.

Steps 7 & 8)

These steps encompass the acceptance tests and are discussed in the next section.

In addition to these issues one participant of the tests was noted as saying: “Where is my spinning cursor or progress bar indicator thing, that is what I am used to.” While

OpenBSD has never provided for a progress indication facility it seems like a worthwhile addition to the installation environment, maybe for the next version. The next section will discuss the Acceptance Tests of our software.

6.1.4. Acceptance Tests

The acceptance tests were given to all three teams as they were relatively short tests. The purpose of our products is to save time, and as an addendum the need to review log files is a necessity for the automated processes which occur. Our goal was for the groups to be able to analyze the installation logs on the Internet and the patching logs from their email, and then record the number and type of errors in under 30 seconds for each of the installation and patching logs. On average the results were much lower, about 20 seconds, but there are several things to take into consideration about the relevance of this test:

1. The logs did not include any simulated errors which might occur during the process.
2. There existed no summary of the installation logs or to view the logs by the severity of entries. There might be a possibility for a single error message to become lost among all of the other entries.
3. Paranoid system administrators may not be confident in the logging facilities to accurately depict any errors which occur so they might check the validity of the installation or patching systems manually.

The patching logs were relatively short and easy to manage, but the above items might be fixed adding further enhancements to both the Internet logging for the installation and email logging for the patching. Additionally the patching logs might be better placed on the Internet also. Furthermore the usability tests did depict a reasonable layout for the configuration files which should assist the users of our product in a speedy alteration of the configuration file for their systems.

7. Conclusion

We believe that, in this project, we have created two projects which have a wide range of application not only for our sponsors but for the international OpenBSD community. We developed the projects with an eye for this larger scope, and, while time has yet to tell whether our installer and patcher will become common OpenBSD tools, we are confident that the community will recognize their value.

The project went very well and, while there were some rough spots and we were not able to achieve all the functionality we would have liked (a few applications such as TFTP support were abandoned), we are proud of the work we have done and happy to have our names attached to this work. Our sponsor is satisfied with the final product, and we eagerly await review from the OpenBSD community.