

Computer Science Department
CPSC 097

**Class of 2006
Senior Conference
on Intrusion Detection**

Proceedings of the Conference

Fall 2005
Swarthmore College
Swarthmore, Pennsylvania, USA

Order copies of this proceedings from:

Computer Science Department
Swarthmore College
500 College Avenue
Swarthmore, PA 19081
USA
Tel: +1-610-328-8272
Fax: +1-610-328-8606
kuperman@cs.swarthmore.edu

Introduction

About CPSC 097: Senior Conference

This course provides honors and course majors an opportunity to delve more deeply into a particular topic in computer science, synthesizing material from previous courses. Topics have included advanced algorithms, networking, evolutionary computation, complexity, encryption and compression, parallel processing, and natural language processing. CPSC 097 is the usual method used to satisfy the comprehensive requirement for a computer science major.

During the 2005-2006 academic year, the Senior Conference was led by Benjamin A. Kuperman in the area of Intrusion Detection.

Computer Science Department

Charles Kelemen, Edward Hicks Magill Professor and Chair
Lisa Meeden, Associate Professor
Tia Newhall, Associate Professor
Richard Wicentowski, Assistant Professor
Benjamin Kuperman, Visiting Assistant Professor

Program Committee Members

Daniel Crosta
Heather Jones
Ethan Jucovy
Benjamin Kuperman
Connie Li
Alan McAvinney
Taufik Parsioan
Kenneth Patton
Javier Prado
Benjamin Turner

Table of Contents

<i>Nectar: A Browser-Agnostic Contextual Web Annotation Tool</i>	
Dan Crosta	1
<i>Building a Heterogeneous HoneyNet</i>	
Javier Prado and Heather Jones	9
<i>Profiling HoneyNet Attackers</i>	
Connie Li and Taufik Parsioan	19
<i>Building a Neural Network for Misuse Detection</i>	
Alan McAvinney and Ben Turner	27
<i>Rabbitstew: A Robot Simulator with Variable Morphologies</i>	
Ethan G. Jucovy	34
<i>Sweeter HoneyNets</i>	
Kenneth Patton	45

Conference Program

Tuesday, December 6, 2005

- 2:45-3:15 *Nectar: A Browser-Agnostic Contextual Web Annotation Tool*
Dan Crosta
- 3:15-3:45 *Building a Heterogeneous HoneyNet*
Javier Prado and Heather Jones
- 3:45-4:15 *Profiling HoneyNet Attackers*
Connie Li and Taufik Parsioan

Thursday, December 8, 2005

- 2:45-3:15 *Building a Neural Network for Misuse Detection*
Alan McAvinney and Ben Turner
- 3:15-3:45 *Rabbitstew: A Robot Simulator with Variable Morphologies*
Ethan G. Jucovy
- 3:45-4:15 *Sweeter HoneyNets*
Kenneth Patton

Nectar: A browser-agnostic contextual web annotation tool

Daniel S. Crosta

December 14, 2005

Abstract

Nectar enables users of the MoinMoin collaborative wiki editing environment to conveniently create and view in-text annotations to wiki documents without disturbing the underlying document. It is designed to facilitate online collaboration for traditional classroom learning by allowing authors to lock documents for editing, but still solicit comments online. Nectar has an obvious and easy to learn yet unobtrusive interface, supports arbitrary threaded conversations started from any annotation, and supports both contextual and page-level annotations.

1 Introduction

For both large lecture-style classes, and smaller discussion-oriented ones, the World Wide Web has proven to be a valuable resource, allowing both teachers and students to extend learning beyond the walls of the classroom. Early, primarily commercial work (eg Blackboard [3]) focused on easing course management for large classes, with tools like gradebooks, online document submission and distribution, and self-grading quizzes. Recently, focus has begun to shift toward supporting collaboration among students in small classes and groups within larger classes on the web. Most educational

courseware systems now include at least online discussion forums associated with each class, and some support collaborative editing environments (eg Blackboard's "Team Sites" tool) which allow users to create web documents quickly, without being bothered to learn complex markup languages.

These tools, however, are not tightly integrated in the online interface, so that discussion boards and collaborative editing tools are often a few clicks away, on a different web page. I believe that for collaborative tools to be widely adopted, they must be available in all parts of the online environment. This allows discussion and collaboration around web documents to take place in context, as discussion of paper documents in class takes place with the documents in front of participants.

One area of active industrial and academic research hopes to bridge the gap between traditional in-class and web learning by enabling students and faculty to create in-text annotations on internet documents. This is usually called web annotation. In particular, Chong and Sakauchi [6] report that "students have the tendency to write everything down during a lecture...for fear of missing something important." As laptop and tablet computers become more widespread, it will be possible for students to have access to prepared lecture notes during the

lecture. Web annotation, then, allows students to make their own notes directly “on” the online lecture notes, instead of having to keep personal notes and prepared lecture notes separate, avoiding the “split interface” problem also identified by Chong and Sakauchi.

Web annotation is not a new idea, and several commercial and academic groups have made preliminary stabs at it. Each implementation tackles part of the overall goal, with varying degrees of success. This work attempts to draw together insights and advances made in the past eight years, and create an integrated web annotation system for the MoinMoin [12] online collaborative Wiki [10] environment. The Wiki environment, while not specialized for educational use, is a promising platform nonetheless, as its flexibility allows for a variety of uses. I believe that an annotation tool in this environment is more valuable than one specialized for any particular existing courseware system, since it need not be specialized for all the types of documents one might want to annotate.

2 Related Work

As early as 1997, instructors at Freiburg University identified the need to allow students to make annotations to online course material. Bacher et al. [2] describe a case study of moving course materials for an algorithms and data structures class to the world wide web. In it, they identify the need for an analog to marginal notes students would make on in-class handouts. Additionally, they identify the need for “annotations at various levels,” where notes may refer to either an entire document, or some paragraph or segment of text from within the document. This notion was later termed “idea-level placement.” [5] Un-

fortunately, they do not describe algorithms nor data structures for dealing with changing underlying document text. In this system, notes are primarily for personal use, though they can be shared on demand.

Cadiz et al. [4] performed a case study of a working online annotation system at a business with over 400 users and over 1,200 documents, in which they discuss the “orphaning” problem of notes losing the context to which they originally refer. To address the problem, they suggest automatic notification to the note’s author, who may then choose new context for the note, or determine that it may no longer be relevant to the document.

The SCHOLION [9] system is a distance-learning solution for publishing lecture slides. Students may arbitrarily annotate these slides for personal or shared use. Additionally, students may reply to questions or notes posted by other students by creating links from annotations to a class discussion board area. It is unclear whether the discussion thread contains a reverse link to the original annotation.

Finally, Chong and Sakauchi [6] present the first in-browser web annotation tool geared toward educational needs. Among their design goals, they reiterate the study benefits of annotating course materials. They also wish to support students working in groups either concurrently or asynchronously, both for collaborative projects and group review sessions.

3 Overview and Goals

I believe that a next generation web annotation system ought to support all of the following features, from an end user’s perspective:

- *Use context when it makes sense.* Previous work [2, 4, 6, 9] has shown that for many tasks, notes should be created and displayed in context. If a note refers only to a particular sentence, that should be made obvious in both input and display. On the other hand, some annotations may refer to the document as a whole (imagine, for example, concluding thoughts a professor makes on a student’s paper). Thus, we have two annotation types: *idea-level* and *page-level* annotations.
- *Annotations are conversations.* SCHOLION [9] introduced the notion of linking discussions to annotations. In the Microsoft Office Web Discussions system discussed in [4], this idea is taken further, by putting the conversation in context within the document being discussed. Additionally, the notes should be viewable as conversations in their own right, with *backlinks* to the document which spurred the conversation. A web annotation system should not impose one interface metaphor or the other on users; both should be available.
- *No learning curve.* Using the annotation system should not be an imposition on the user, either by requiring the installation or use of special software (eg a new browser, plugin, or other software), nor with a clumsy or unusual interface. Using standardized web technologies ensures that users will not be confused by the interface, and has the added benefit of supporting nearly all users without any modification to their computer systems. Nectar requires only a CSS- and JavaScript-capable web browser, which is a safe assumption on nearly every com-

puter from which a web annotation system is likely to be used.

These goals, then, inform the design of the Nectar system. The most obvious data structure with which to implement threaded notes is a tree, in which each branch represents a thread of conversation. No restrictions are placed on the branching pattern of the tree. Some notes may comprise only a single node (if there are no replies), while others may spawn deeply threaded discussions, which will take the form of highly branching or very tall trees. Conceptually, the document itself forms the root node, of which each annotation is a child. Replies to annotations form additional leaf nodes, which can become internal nodes when they themselves are replied to.

Idea-level notes store an additional field in the root node: the location in the document that the note references as context. This is used to locate the idea-level notes in the text when displaying the document, and also to gather the context for display alongside the annotation.

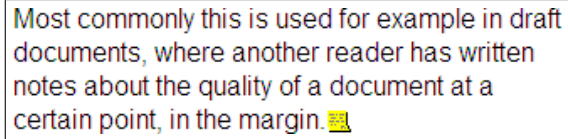
4 Implementation Details

The Nectar system is implemented as a set of plugins to the MoinMoin wiki system. The wiki, with its emphasis on shared authorship of web documents, is a solid starting point for developing a collaborative educational courseware system. In addition, MoinMoin handles many of the details of a multi-user online editing environment, such as user authentication and access control. MoinMoin’s modular design also allows the wiki formatting markup to be reused within the annotation plugins.

Storage Notes are stored on the web server as XML documents, with one note per file. MoinMoin has a separate directory structure for each page of the wiki, in which Nectar creates a `notes/` subdirectory, so notes are stored, as well as displayed, contextually. This helps offset some of the cost of using the filesystem to store notes, since it is known exactly which set of notes apply to any given document. Each file has a unique filename, generated from the creation date, type, and title of the annotation it contains. The filename type marker further defrays the cost of using the filesystem, as notes need not be read to determine which type of note it contains. The tree structure is a doubly-linked tree, in which each root and internal note contains a list of child note filenames, and each internal and leaf note contains the filename of its parent. This makes it possible to recursively reconstruct the entire thread of conversation from any note, which would be helpful when an internal or leaf note is found as a result of a search.

Storing each note as a file makes the procedure for displaying notes with a document simple (list the `notes/` directory) rather than complex (creating or dealing with a more robust database system). It does not, however, require significant wasted effort to compute note threads, since identifying root nodes requires only listing the directory, rather than reading and parsing the note files themselves. Once the root note is parsed, all the information necessary to create the next level of the tree is present in that file. For this reason, it was only more complex, and not significantly more efficient, to store notes using a hierarchy of folders and files to mirror the tree structure, rather than storing all notes in a single directory.

Storing notes on the filesystem as XML carries with it some overhead in space and time which



Most commonly this is used for example in draft documents, where another reader has written notes about the quality of a document at a certain point, in the margin. 📌

Figure 1: The note icon indicates the presence of an idea-level annotation.

may be unsuitable for large deployments. In particular, since each note is stored in its own file, small notes take up more space on disk than they otherwise might, since the minimum actual file size in most filesystems is the block size (often four kilobytes). The average note, on the other hand, even including the verbose XML wrapper, is usually under one kilobyte. In a relational database, the note would only take up as much space as its contents and metadata (creation date, type, context region, etc) require, and the block size issue is not encountered since all database records are typically stored in a single file. Additionally, searching notes stored in the filesystem would require opening all the files and parsing the XML, which is probably significantly slower than search in a relational database. Implementing acceleration techniques (eg indexing note contents) duplicates much of the effort put into developing database systems. Nectar's storage and interface implementations are kept strictly separate, to facilitate adapting Nectar to use a more robust storage system, should this need arise.

Rendering During page rendering, all the root notes for a page are loaded and parsed. For each idea-level note, an icon is placed in text at the end of the referenced area, to indicate that a note exists. When the user clicks the note icon (Figure 1), JavaScript [7] running in the browser modifies the page's Document Object

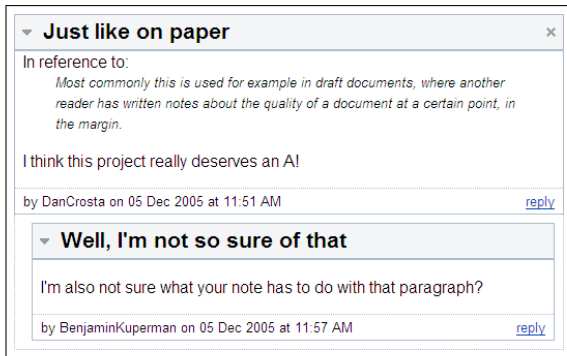


Figure 2: For idea-level annotations, this box displays the threaded conversation floating “above” the document text. The triangle and × icons may be used to collapse or completely hide the notes, respectively. General annotations are displayed similarly below the document text.

Model (DOM) to “pop up” a box in the same page to display the notes (Figure 2). Page-level notes, are displayed at the bottom of the page, and are always present when viewing the page. Both idea- and page-level notes are displayed with a collapsible, threaded view, so that the user may choose which notes and replies to view at any given time.

Creating Notes To create a note, the user has several options. To create new page-level notes, the user fills out a form at the bottom of the page, prompting for a title and comment about the page. Creating an idea-level note is as simple as selecting a portion of the document and hitting the “n” key (for **n**ew or **n**ote, whichever is easier to remember). When a user does so, JavaScript again modifies the DOM to pop up a form similar to that of a page-level annotation, which the user may fill out to create a new idea-level note. The form also displays the text the user selected, for verification (Figure 3). The forms are submitted to the webserver, which up-

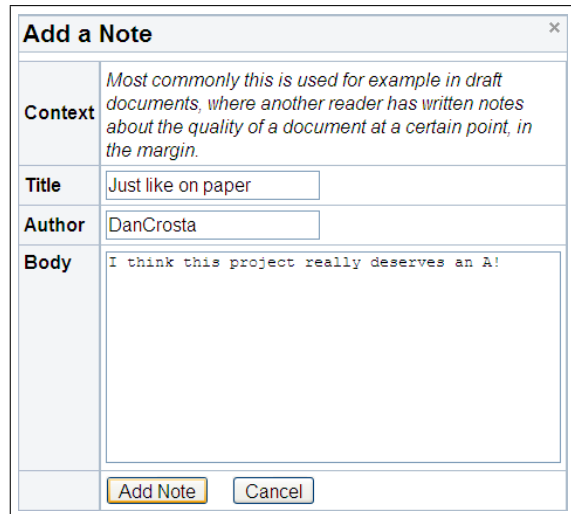


Figure 3: To create new annotations, the user is presented with this prompt for information. The form for creating a new general note is similar, but lacks the “Context” display area.

dates the note tree data structures, and then re-displays the page to the user.

Detecting the range of text a user selects is accomplished through a bit of JavaScript and CSS [14] trickery. At page render time, the following snippet of HTML code is inserted into the middle of each word in the document:

```
<span class="nw">_nw_N_</span>
```

where N is a running word count in the document. Ordinarily, all the text in the `span`, that is, the `_nw_N_` would be displayed, but the CSS stylesheet loaded for the page disables this display. However, since the markers are part of the page text, the JavaScript `getSelection()` method used to find what text the user has selected will contain them. Special care is taken when inserting the invisible markers not to in-

validate the XHTML [15] structure of the document, so that it displays properly in all browsers.

A regular expression is used to parse out the first and last markers in the selected text, and these are sent to the server along with the user's form submission. Since the markers are inserted into the middle of each word, the first and last markers are the word indices of the first and last words, respectively, to be more than half selected by the user. A little additional parsing is required to handle single-letter words, but this is straight-forward and not prone to mistakes.

This hidden marker system is not without costs, though. Since the marker itself is significantly longer than the average word length, for long documents (where the length of the document text outweighs the static overhead of wiki headers), the hidden markers may add as much as five to six times more data to transfer. Additionally, the markers are visible in the document source, which may be undesirable to some users. Unfortunately, this is the only system likely to work across the diversity of web browsers currently in use. There are several possible solutions to these problems, discussed in detail in Section 5. To decrease overhead, if a particular user does not have permission to annotate a given document (permissions in MoinMoin documents are set by document editors, in the form of access control lists), none of the hidden markers are sent to the user's computer, nor does the interface for adding annotations (by highlighting) appear.

5 Future Work

The problem of changing underlying documents, identified earlier by [2, 4, 6], remains to be solved. Though Nectar currently does not imple-

ment a solution, MoinMoin implements a form of version control similar to that provided by CVS [8]. In particular, changed portions of the document can be identified between any two revisions. With this information, it may be possible to preserve many annotations in changing documents by updating the context markers according to how the document changed. It may also be possible, though more complex and expensive, to use language processing techniques to relocate annotations with a high degree of success based on the contents of the document itself. As a fallback, users should be notified when the document is changed, as suggested in Cadiz et al., so that they may update or invalidate their annotations as appropriate. Currently, Nectar makes the simplifying assumption that documents do not change, even though this is not realistic.

Another major problem, which seems specific to Nectar, is the extra overhead of the hidden markers required for idea-level annotations. Some existing systems [4] allow annotations only at predefined intervals, eg the end of each sentence or paragraph. I believe that word-level granularity or finer is necessary to create a useful analog to pen-and-paper annotation. Another possible solution is to rely on JavaScript to insert the hidden markers in the plain text after it is delivered to the browser. However, this requires a consistent implementation of regular expressions in all browsers, which has not been the case in my experience. Finally, the extra overhead of the hidden markers may become insignificant even in long documents as high bandwidth internet access continues to expand.

Other features that might be added to Nectar are different interaction modes with the annotations. In particular, further integration of the discussion metaphor with the annotations would be ideal. Users should be able to browse anno-

tations either through the document annotation interface Nectar currently implements, or a “forums” view, where annotations are presented as a list of conversations about a wiki page, with appropriate backlinks to the original document. A feature commonly found in web forums is search, which also makes sense for annotations. Both of these features should be straightforward to implement in future versions of Nectar.

6 Post-Mortem Reflections

When I began this project, I wanted to recreate an interface similar to Microsoft Word’s [11] “track changes” interface. Unfortunately, the diversity of browsers, all implementing different subsets of different interfaces and technologies, and none of which are geared toward designing interactive user applications, makes this goal nearly unattainable. I ended up doing the lion’s share of the work on the server side, for the goal of consistency between browsers and ease of implementation. It is, in many cases, possible to create fully cross-browser JavaScript applications with advanced functionality, but in most cases this requires creating a meta-API where each function your application calls is a wrapper which first detects browser capabilities, and then decides how best to go about the task. Developing such a meta-API requires time I did not have this semester, so at this point, Nectar is only tested and known to work with Mozilla Firefox [13] and Apple’s Safari [1]. In future versions, I expect that more of the work can be done on the client side, easing the load on the server and increasing Nectar’s scalability.

When comes to the actual code of Nectar itself, I am now beginning to understand the “write it once, throw it away, write it again”

mentality in software design. There were some challenges I didn’t see ahead of time, in particular, the need for inserting hidden markers in the text. I thought that it was generally possible to determine which DOM nodes were selected by the user, or in which DOM nodes the selection was, which ought to have been enough to figure out selection from the plain document text. This stumbling block alone was enough to delay Nectar by about a week as I struggled first to find out how it was possible (“it *must* be possible, right?”), and then to figure the best way around it. There were a few ways to insert the hidden markers, and it took some time to decide on the middle-of-the-word strategy. This seemed to represent the best mix of minimizing overhead while still allowing flexible and useful annotation.

I also found the MoinMoin API a little messy and undocumented, and in particular often got frustrated at the “TODO: update this comment for 1.3.5” type comments strewn liberally throughout the code. I think free and open source software is a fantastic idea, and the availability of MoinMoin’s source code made this project possible, but I wonder if it is possible in such an organically grown project as MoinMoin to maintain strict standards of coding, testing and documentation.

Finally, I wish that neither of the above had been problems so that I could have had time to tackle what I see as the major advance of Nectar among web annotation systems, the “forums” interface I mentioned in Section 3. It has always been a priority for me, both in this project and in other applications I have made, not to try to impose interaction methods on the user, or at least to avoid doing so as much as possible. In addition to adding interface flexibility, I had hoped that the forums interface would allow users to

discover new wiki pages by browsing what was being said about them. It has certainly been my experience in education that the discussion around and about class readings, for instance, is both more in-depth and more comprehensible than the primary document; I suspect this is true for other fields, as well.

References

- [1] Apple Computer, Inc. Apple - Mac OS X - Safari RSS. URL <http://www.apple.com/macosx/features/safari/>.
- [2] C. Bacher, R. Müller, T. Ottmann, and M. Will. Open hypermedia educational environments: A feasible approach to overcome some difficulties. Technical Report 91, Freiburg University, 9, 1997.
- [3] Blackboard, Inc. The Blackboard Academic Suite. URL <http://www.blackboard.com/products/as>.
- [4] J. J. Cadiz, Anop Gupta, and Jonathan Grudin. Using web annotations for asynchronous collaboration around documents. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 2000.
- [5] Laurie Causton. Web-based tools for document annotation, 1999. URL http://www.e1pub.org/html/tool_annot.html. Document no longer available, cited in [6].
- [6] Ng S. T. Chong and Masao Sakauchi. Creating and sharing web notes via a standard browser. *ACM SIGCUE Outlook*, 27 (3), September 2001.
- [7] ECMA International. Stanrard ECMA-262: ECMAScript Language Reference, December 1999. URL http://www.ecma-international.org/publications/ECMA_ST/Ecma-262.pdf.
- [8] Free Software Foundation. CVS - Open Source Version Control. URL <http://www.nongnu.org/cvs/>.
- [9] Barbara Froshcauer, Chris Stary, Michael Ellmer, Thomas Pils, Wolfgang Ortner, and Alexandra Totter. Scholion: Scaleable technologies for telelearning. In *Proceedings of the 2000 ACM symposium on Applied computing*, volume 1, 2000.
- [10] Bo Leuf and Ward Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley Professional, April 2001.
- [11] Microsoft Corporation. Microsoft Office Online Home Page. URL <http://office.microsoft.com/en-us/default.aspx>.
- [12] MoinMoin. URL <http://moinmoin.wikiwikiweb.de/MoinMoin>.
- [13] The Mozilla Corporation. Firefox - Rediscover the Web. URL <http://www.mozilla.com/firefox/>.
- [14] World Wide Web Consortium. Cascading Style Sheets, level 2 revision 1 CSS 2.1 Specification, June 2005. URL <http://www.w3.org/TR/CSS21/>.
- [15] World Wide Web Consortium. XHTML 2.0, May 2005. URL <http://www.w3.org/TR/2005/WD-xhtml2-20050527/>.

Building a Heterogeneous Honeynet

Javier Prado
jprado1@swarthmore.edu

Heather Jones
hjones2@swarthmore.edu

December 13, 2005

Abstract

A small network of honeypots, each running a different operating system, was constructed inside the campus network at Swarthmore College. Although preexisting honeynet software was used, quite a few difficulties were encountered. These problems, and solutions to those that were solved, are presented in this paper.

1 Introduction

It is critical for any misuse intrusion detection system to know how to be able to detect the latest attacks. With new attacks emerging every day, this can be quite difficult. One tool developed to assist in keeping pace with the attackers is a honeypot: a computer with no purpose other than to be attacked and collect data on the attacks it suffers. A honeynet is an entire network used in this way. Because so many of the attacks in use today involve an intruder gaining control of a computer and then using it to attack other machines, a honeynet operator must make sure that his honeynet cannot only collect data effectively, but can also stop his machines from being used to launch further attacks. This makes the task of building a honeynet from scratch very difficult. Fortunately, there are a number of tools already available to capture and control traffic on a honeynet. The most important contribution, perhaps, is the Honeynet Project's Honeywall Roo, which combines several honeynet technologies onto a bootable CDROM, including the network packet sniffer Snort, a server for the host-

based data capture module Sebek, the ability to limit traffic from the honeynet, and a web GUI for administration and data analysis. The idea of the Roo CDROM is to make it easy for any organization to set up its own honeynet.

We set out to build a honeynet using the Honeywall CDROM Roo in order to research the different kinds of attacks that occur on different operating systems. We found, however, that setting up a honeynet is not as easy as it looks. Some of our problems were due to bugs in various parts of the system. Others occurred when we failed to understand key details about how the system was supposed to work. While the latter could be dismissed as due entirely to our inexperience, we believe they provide valuable information about what the makers of the CDROM Roo have assumed that their users will know.

First, we will provide some background about the Honeynet Project, other research on attacks, and details of how the Honeywall Roo works. Next, we will discuss the architecture of our honeynet and the problems we encountered with some of the components of the system. Finally, we will draw conclusions about what we have learned from this experience.

2 Background

2.1 Honeypots and Honeynet.org

Beginning in 1999, the Honeynet Project has been developing and using honeynets to collect and distribute knowledge about the black-hat community [6]. The first phase of the project

involved testing the idea that data captured by honeynets could provide useful information about attacks. In the second phase, which began in 2002, a second generation of honeynet technology (Gen II) was developed to be simpler, more interactive, safer, and easier to deploy than previous architectures. The third phase packaged everything necessary to set up a honeynet onto a bootable CD-ROM. This has made it much easier for organizations around the world to set up their own honeynets. The fourth phase involves the development of a centralized data collection system. As the project has progressed, it has expanded to become the Honeynet Research Alliance, with 20 member groups around the globe.

We are using the Honeynet Project's Gen III technology to implement our honeynet. Significant improvements made between Gen II and Gen III include a more automated installation, a web GUI for administration, and a capability for automatic updates [9].

A similar experiment to the one we initially set out to do here was performed as a part of the Honeynet Project. They compared data from Windows, Linux, Solaris and OpenBSD machines, and discovered that each operating system attracted different kinds of attacks. On Windows systems, they saw worms and automated attacks. On Linux systems, they saw attacks originating mainly from Eastern Europe, especially from Romania, that exploited known vulnerabilities or employed automated tools. On Solaris and OpenBSD, they saw more advanced or interesting attacks [6].

2.2 Related Work

In a 2005 paper entitled "A Pointillist Approach for Comparing Honeypots," Pouget and Holz examined how a three-machine honeynet running Windows 98, Windows NT Server and Redhat Linux 7.3 server could be used to make a low-interaction honeypot emulating these three machines more believable [2]. They classified attacks into three categories: Type I attacks targeted only one machine, Type II attacks targeted

two out of three machines, and Type III attacks targeted all three machines. Approximately 60 percent of the attacks they recorded were of Type I, and 35 percent were of Type III. Of the few Type II attacks, 88 percent were judged to be Type III attacks for which the message to one of the machines had been lost, 9 percent were due to scanning attacks that targeted every other IP address, and 3 percent were believed to be attacks on the two Windows machines only.

Yegneswaran, Barford and Ullrich used data from 1600 networks around the world to study the global characteristics of internet attacks[11]. In addition to analyzing the volume and distribution of attacks in their 2003 paper "Internet Intrusions: Global Characteristics and Prevalence," they presented a classification of four scan types. A vertical scan examines several different ports on a single machine. A horizontal scan examines the same port on several different machines. A coordinated scan examines the same port on the machines within a subnet and originates from several sources. A stealth scan is a horizontal or vertical scan executed with a very low frequency in order to avoid detection.

Weaver, et al. develop a taxonomy of computer worms based on method of target discovery, carrier, activation, payload, and attacker motivation in their 2003 paper on the subject[10]. Following this taxonomy, target discovery techniques are scanning, pre-generated target lists, external target lists, internal target lists and passive discovery. Carriers include the worm itself, a second channel through which the worm completes an initiated infection, and normal traffic in which the worm embeds itself. A worm can be activated by a human, by a human activity, by a scheduled process, or by itself. The most common payload for a worm is a nonexistent or nonfunctional one, although there are many other possible payloads, including electronic or physical remote control, damage, or denial of service. Attackers may be motivated by experimental curiosity, pride, commercial advantage, criminal gain, random protest, political protest, terrorism, or cyber warfare.

2.3 Honeynet Details

The Know your Enemy series of documents made available by the honeynet.org website has proved to be an invaluable resource. The seminal paper for any honeynet project is the Know your Enemy: Honeynets paper [8]. This paper gives a survey of the basic, fundamental concepts of a honeynet. The two fundamental aspects of a honeynet as they apply to our project are the data control concept and the data capture concept. In data control, the key concept is that it is imperative to circumvent an intruder's ability to attack other systems outside of the honeynet once the intruder has compromised a honeypot within the honeynet. This responsibility primarily rests on the honeywall's ability to detect malicious activity on the network and in each of its honeypots, and then respond in turn to the threat at hand. The paper makes clear that the actual implementation of the data control (our honeywall) is ultimately our decision but the author does provide some suggestions for our implementation. Among the suggestions include the layering of security measures to help obfuscate the presence of the honeywall and for the honeywall to operate in a fail closed manner where any failure of one of our components will result in all network traffic from the honeynet being terminated [8]. The other fundamental concept of a honeynet, data capture, concerns the logging and reporting of an intruders activity, basically the reason why the honeynet exists. For our project, the two primary elements of data capture are the Sebek client and the honeywall's log of network activity.

Because the main goal of this project was to get the honeywall to the point where it could collect useful data, the Sebek client is one of the fundamental components of our project. Sebek is a solution to a problem that has faced the honeynet community: how to observe an intruder's actions without the intruder knowing that he or she is being monitored. The paper discussing this issue [7] shows how simple network monitoring, although a viable solution, is undermined by

an intruder's ability to use encryption to obfuscate his or her activity. The only way to capture unencrypted data is to catch it before it is encrypted (i.e., before the attacker sends it onto the network), or after it is decrypted (i.e., once it has reached its final destination). Thus, Sebek provides information on an attacker's actions from the attacked host. In this way, data on the attacker's keystrokes, processes run, files opened, and other system activities can be recorded. The key feature of the Sebek client is that it is incorporated into the kernel of the operating system¹. Since it is a part of the kernel, it is able to effectively hide from the intruder. Conceptually, this article [8] describes that the user space and the kernel space are mutually exclusive. This being the case, having the Sebek client become a part of the kernel is a solid way of hiding the data capturing component of a honeynet. The Sebek client then stores in a buffer the recorded actions of an intruder, encrypts the information, and sends it to the honeywall to which the honeypot is connected. Because the Sebek client is installed on the attacked host, it provides information about what happens on the host which could not necessarily be gathered from an examination of the network traffic to and from the host, even if this traffic could be decrypted. Thus, Sebek provides a valuable complement to the network data collected by the honeywall[7].

While making the Sebek client a part of the operating system helps to keep it hidden from the attacker, it also creates a problem when honeypots with different operating systems are used. As Windows is even more different from Linux than FreeBSD is from Solaris, we need to ensure that each Sebek client we install is appropriate

¹Sebek was originally adapted from a blackhat rootkit called Adore [1]. Such rootkits were developed to change the behavior of the kernel without revealing their existence. Recently, Sony has developed a rootkit aimed to enforce usage rules on copyrighted materials [3]. This has produced sharp criticism, not only from advocates of privacy, but also from Windows experts who claim the code is poorly written and can compromise the security of the Windows operating system.

for the operating system that it is going to be installed on. The Honeynet Project has completed much of this process of adapting the Sebek client to each operating system environment. For the Windows environment, the developers have gone so far as to provide a simple executable which automatically incorporates the Sebek client into the kernel of the Windows operating system and leaves the incorporated Sebek client to be customized by the user. For FreeBSD, a loadable kernel module of the Sebek client is available. For Linux, the kernel must be rebuilt to incorporate Sebek. Information regarding the actual Sebek client implementation exists on the Sebek homepage within the honeynet.org website [4].

3 Our Own Honeynet

We wanted to examine the differences between the attacks seen on different operating systems, so we decided to set up a Honeynet with several computers running different operating systems. We chose to use the Honeynet Project's Honeywall Roo bootable CD-ROM, believing that this would make the setup portion of the project virtually effortless. Unfortunately, this did not turn out to be the case. Instead, we found that the setup became a project in itself. What follows is a description of our honeynet, the problems we encountered, how we solved these problems, and what still needs work.

3.1 Honeynet Architecture

Our honeynet consisted of five machines: one running the Honeynet Project's Honeywall Roo (including a Fedora core), one running RedHat Linux, one running Windows XP, one running FreeBSD, and one running Solaris. Because there was no version of Sebek for Solaris that was compatible with the Gen III honeywall, we planned to simply examine the traffic going to and from the machine. After weeks of crawling around behind our computers plugging and unplugging various cables, we obtained a KVM

switch to allow us to easily interact with any of the machines except the Solaris using the same monitor, keyboard and mouse. Our honeynet was located within Swarthmore College's campus network; we will refer to this external network as the internet, although being inside of this network did present problems that we would not have seen had we actually been directly connected to the internet. All the individual honeypots were connected to the honeywall through a LanTronix 10 base-T ethernet switch (See Figure 1).

3.2 The Honeywall

The honeywall was a Dell Precision 330 with an Intel Pentium 4 processor. We installed two additional network cards, allowing us two ethernet connections to the internet one as a route to the honeypots and one for remote management of the honeywall and one ethernet connection to our own honeypot network.

We set up the honeywall to run in bridge mode as opposed to NAT (Network Address Translation) mode (See Figures 2 and 3), allowing each honeypot to have its own IP address because we wanted each to appear as a distinct target. We initially had only two network cards because the Honeywall documentation indicated that a third network card was only necessary if remote administration was desired. Yet, this simple setup was not functional because our honeywall software kept trying to find a third network connection. This prevented us from completing a full installation of the honeywall, so we installed a third network card. After this installation, we needed to provide an IP address for honeywall's configuration utility. Since we were behind Swarthmore College's network, we could not simply set an IP address arbitrarily. Because the honeywall installation did not include a DHCP client or any other similar IP configuration program, we could not determine the IP address from the honeywall machine. We solved this problem by using the live Linux CD Knoppix. This allowed us to run dhclient to get an

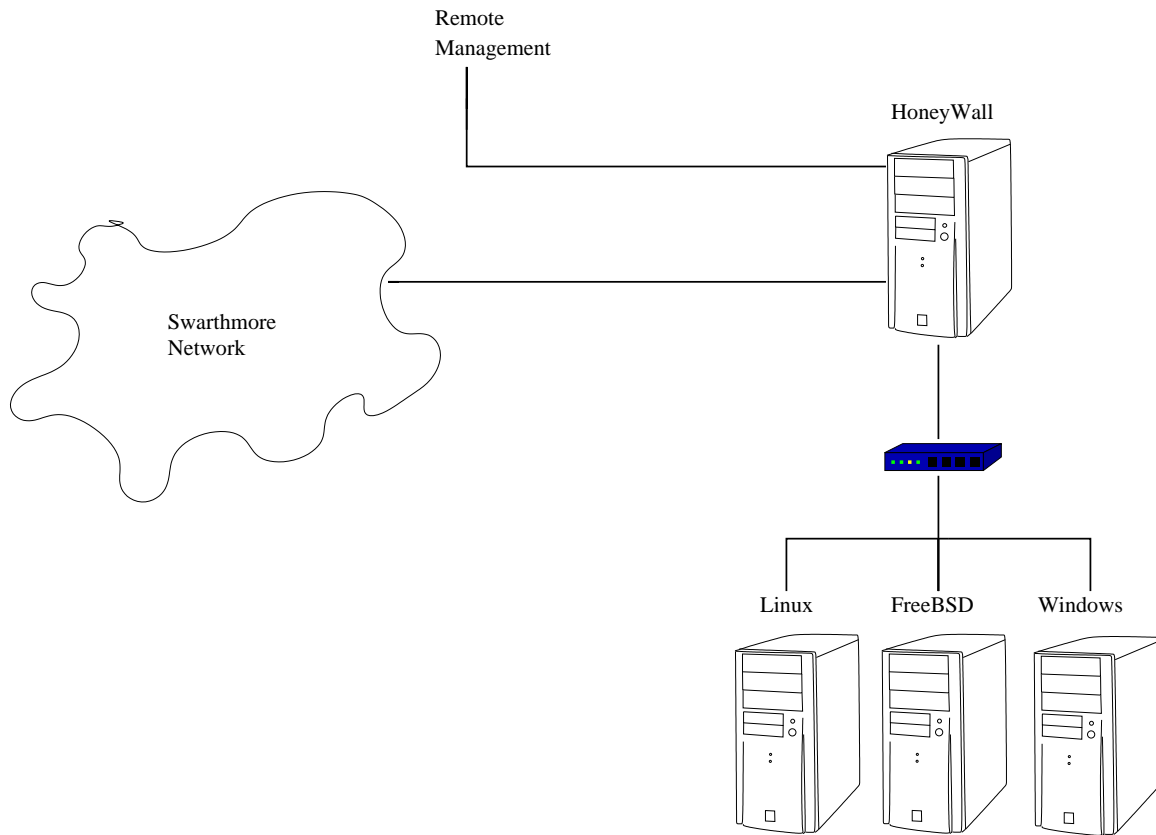


Figure 1: Diagram of Honeynet Architecture

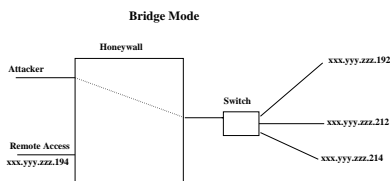


Figure 2: Diagram of bridge mode configuration

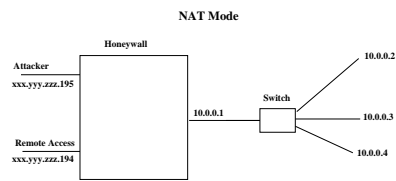


Figure 3: Diagram of NAT mode configuration

IP address which we could then enter into the configuration utility.

One of the main improvements that came with the Generation III honeywall technology was the web interface, Walleye. This GUI allows the honey-net administrator to interact with the honey-

wall in a more user-friendly environment than the text-based menu utility that can be run on the machine. Using this interface, we were able to observe a small amount of traffic on coming to and from our honeypots mostly bootp, domain or NetBIOS traffic, plus any traffic we created

ourselves while testing the system. Supposedly, Walleye identifies what operating system each machine is running. While this occurred successfully for some of the machines connecting to our honeypots, it initially did not work for the honeypots themselves for some unknown reason.

On two occasions, we found that we could not log in to Walleye at all. After we entered in a username and password, the system would begin the login process but load so slowly that it never got to the welcome page. We then logged into the honeywall directly to investigate what might be wrong. Reloading the or even rebooting the honeywall did not fix the problem. Eventually, we realized that the directory where the honeywall was storing its data had become full. The first time, moving the data to a directory with more free space fixed the problem. The second time this problem occurred, we realized that we needed more space than was available on the honeynet's hard drive, so we dumped all the data to another machine. We then used the menu utility to clean out the logging directories. In doing so, we noticed an interesting message: apparently the honeywall had not yet been programmed to clean out its MySQL database files. Since the honeywall was still failing to start MySQL, which it uses to make its collected data available to view over the internet and manage the users for Walleye, we initially tried removing several database files that were particularly large, thinking that perhaps the overly large size of the database was preventing it from loading. This did not fix the problem, however, so we restored the files from the external machine to which we had previously dumped all our data. After re-running the startup script for the MySQL daemon, we investigated the error produced. The error itself was not very informative, but it did point us to a log file, which indicated that one of our database tables may have been corrupted. A quick Google search on repairing MySQL tables revealed the program `myisamchk`. This successfully repaired the table, but then the startup script complained about the next table.

After running the program on all of the tables in the database, we were finally able to start the MySQL daemon and successfully log in to Walleye. Looking back at this problem, we found it surprising that the amount of data generated was large enough to cause significant issues with the honeywall machine, since we saw only background noise and test traffic, and no actual attacks. Clearly, for a honeynet that is being used to collect data for research, even for a relatively short term project, the issue of where and how old data should be stored would have to be very carefully considered.

The simplest way to examine traffic on the honeynet is to search for connections by date, time, IP address, and or port from the Walleye welcome page. The results can either be returned as a PCAP file or in "Walleye flow view," the latter being much easier to interpret. At the beginning of the month of December, however, we began encountering an error whenever we tried to see results in Walleye flow view. The error stated that the month "12" was out of the range 0..11, and indicated a file on the honeywall where the problem had occurred. An examination of this file was not enlightening, as it was written in Perl script, with which we have no experience. A search of the Bugzilla bug information server on Honeynet.org produced no results. We tried running YUM (Yellow-dog Updater, Modified) again to get the latest updates for the honeywall, but the only change we observed was that the graph of recent activity on the honeywall, usually displayed on the Walleye welcome page, was no longer loading correctly. By this time, we were several days into the month of December, and another search of the Bugzilla server produced one record. This linked to a possible solution: change one number in the file on the honeywall that had been causing problems [5]. We tried this, and sure enough, it solved our problem. The author of the tip admitted that he did not know what other effects this change might have, but we have not as yet observed any negative repercussions. We did, however, notice

that in some cases, Walleye could now recognize the operating system of our Linux and Windows honeypots. We are still unsure why this feature only sometimes works: it does not seem to have any correlation to the type of connection being monitored. We also noticed that when we try to see what packets Snort has dropped using Walleye's "System Status" feature, we see a message saying that no packets have been dropped that day, while checking the same field using the menu utility does display information on a number of packets that have been dropped.

Since data control is a very important part of any honeynet, we wanted to do a quick test of our ability to control the traffic leaving our honeynet. "Roach Motel mode" allows the honeynet operator to allow traffic in but prevent any traffic out. We tried a test that involved using ssh to log in to a honeypot and then ping a machine outside the honeynet. As expected, this test succeeded fully in normal mode, but in Roach Motel mode the user could log in to the honeypot but the ping command did not succeed. We then examined the "Emergency Lockdown" feature. Supposedly, this prevents any traffic in or out of the honeywall except through the management interface. We found that turning on this option did stop an ssh connection in progress to one of our honeypots, and turning off the option allowed this connection to resume. Unfortunately, all connections to the management interface also failed to work with the lockdown option on. This may be an issue unique to our configuration, since we have our honeypots and our remote management interface on the same Class-B network, (the Swarthmore College network), or it may be a problem for any setup. In either case, it would make a situation where a honeynet is monitored exclusively through the remote management interface impossible.

3.3 Linux

The Linux machine was a Dell Precision 330 with an Intel Pentium 4 processor. We initially had the machine running RedHat 7.2 with a 2.4.9

Linux kernel. The current version of Sebek for Linux 2.4 worked with the 2.4.30 kernel. This posed a problem. We updated the kernel but then we discovered that this updated kernel was not completely compatible with the version of Redhat that we were using. This incompatibility manifested itself in the system's inability to recognize the ethernet card. Since internet connectivity is an essential part of a honeynet, we decided to install a more recent version of RedHat. We installed RedHat 9, updated to a 2.4.30 kernel, and installed the Sebek module. This version of Sebek appears to be functioning correctly: packets are dropped and logged at the honeywall, and examining these packets through the web interface shows that individual commands are being recognized.

We were able to get an IP address on this machine by running dhclient, but we could not ping anything beyond our honeywall or reach any sites with a web browser. We determined that this problem must be due to a setting on the honeywall, but we were unable to locate the relevant field in the menu configuration utility. Then, when examining the configuration page on the Walleye web GUI, we found the source of the problem: the "honeypot public address(es)" field had been set to the IP of the external NAT mode interface of the Honeywall, and the honeywall was not in NAT mode. After entering in the specific addresses of the honeypots, we were able to connect to the network beyond our honeywall.

3.4 FreeBSD

The FreeBSD machine was a Dell OptiPlex GXa with an Intel Pentium 2 processor. We had the machine running FreeBSD 5.2.1. We loaded the FreeBSD version of Sebek 3.0.3 onto the machine, and we have observed packets being sent out, but they are considered by the honeywall to be malformed. The current FreeBSD version of Sebek 3.0.3 is still in testing, so we are not sure whether there is a problem with the program or with the way we have it set up. By running dhclient, we were able to get an IP address on

this machine, however, we soon began getting error messages that this IP address was not unique. Returning several days later, we found an error message claiming that the machine's IP address was 0.0.0.0 and was already in use. Adding the IP address the machine had obtained previously to the `/etc/hosts` file resolved this problem temporarily, but it soon resurfaced. We attempted to run `dhclient` again to obtain a new IP, but the program seemed to be malfunctioning: it ran for several minutes without producing any output. At this point, the network problems of our FreeBSD honeypot are still unresolved.

3.5 Windows

The Windows machine was a Dell Precision 330 with an Intel Pentium 4 processor. The machine was running Windows XP Professional with no service packs or updates installed. We loaded the newly available Windows version of Sebek 3.0.3 onto the machine, and it appeared to be working. When we tried to connect the machine to the internet, however, it began to spontaneously reboot. After a few cycles, it got to the point where it booted up to the login screen and immediately began booting again. We initially assumed this must be due to a virus, but when we examined the network traffic, we were unable to see any connections that may have allowed a virus to reach the machine. We thus concluded that the Sebek client itself was causing the computer to malfunction. We reinstalled the operating system and the Sebek client. This time we saw only a single spontaneous reboot, and after that the machine appeared to function normally. We did see occasional error messages upon login, but we were unable to decipher them, as they were formatted to be sent to Microsoft and not to be readable on the machine. At this point we could see that the machine was sending Sebek packets in response to typed commands, but again the honeywall reported that these packets were malformed. Although this machine did get an IP address, it also could not connect to the internet with a web browser until the afore-

mentioned change to the honeywall configuration (See section 3.3).

4 Conclusion

We have succeeded in building a small honeynet with one fully functional and two partially functional honeypots. Our Linux honeypot collects data using the Sebek client and the honeywall correctly interprets this data. Our Windows honeypot sends improperly formed Sebek packets that cannot be decoded by the honeywall. Our FreeBSD honeypot was previously able to send malformed Sebek packets, but now has problems holding a valid IP address. Our honeywall can limit or fully block outgoing traffic as desired, although a complete system lockdown seems to also block the management interface.

In doing this project, we discovered that the creators of the Honeywall Roo package had made certain assumptions about the knowledge base that their users would have. Those building a Honeynet are expected to know how to set up a network. This may seem obvious, but researchers in a field other than networking, (computer security for example), may have an interest in setting up a honeynet but no previous experience with networks. The honeynet configuration utility also seems to be written for people who are in control of the network on which they are placing their new honeynet. Because we were on Swarthmore College's campus network, this was not the case for us, and we had to briefly run another operating system on our honeywall machine just to get an IP address. In addition, it is assumed that those installing Sebek onto a honeypot know how to work with the kernel of that honeypot's operating system. Installation onto the Windows machine was very simple, although ultimately the installed module did not function correctly. Getting Sebek onto the FreeBSD machine involved inserting a kernel module and stripping the kernel, and this program also did not work as it should have. The Linux version of Sebek also required loading in a kernel mod-

ule. Because the kernel on the machine we had was significantly older than the version that the Sebek module expected, we had to update the kernel - something we had to learn how to do. The instructions for installing Sebek on Linux also mention that the module will be uninstalled every time the machine reboots unless the module is loaded in a startup script. We have never worked with startup scripts, so this is a task that has not yet been completed.

We also encountered problems specific to the honeynet itself. In less than two months, even with no actual attack traffic being observed, the honeynet can fill up the directory it uses to store data. This causes problems with Walleye or any other program that uses the database of captured traffic. In some cases, this can even cause the tables in the honeywall's MySQL database to become corrupted. Running the program `myisamchk` can repair these tables, but this takes a significant amount of time with large tables. Also, the honeywall's menu utility command to clean out the logging directories does not clean the MySQL database files. If these become large enough to cause problems even without being stored in the same directory as other large data files, some method for cleaning them out manually would have to be found. In view of the serious problems that large amounts of data can cause, a honeynet operator should have a location to which old data can be transferred for storage and do this transfer frequently.

The Sebek modules for Windows and FreeBSD produce malformed packets. We are unsure whether this problem is specific to the way in which we have the modules configured on our honeypots or whether it would occur for any configuration. In any case, hopefully the next revision of the module for each operating system will include a fix for this bug.

We encountered a problem in the Walleye interface that prevented it from displaying any traffic for the month of December. This was fixed by making a slight modification to a Perl script file on the Honeynet, (See [5]).

Within the "System Status" section of the Walleye System Administration feature, Snort alerts fail to show up, even if they have occurred and are clearly visible via the menu utility on the honeywall itself. We have not done a thorough investigation of the other options within this section, since most of our focus was on getting the basic features of our honeynet up and working, but this is definitely an area for future work.

During an "Emergency Lockdown," the management interface also appears to be blocked. We are unsure whether or not this problem is specific to our setup, (our management interface is on the same network as the public interface to our honeynet), but it is an important area for further exploration, since it affects whether or not a honeynet can be managed entirely through remote access.

We recognize that the Honeywall Roo is still a relatively new technology, and as such will have many bugs. We hope that this record of our experiences with the software will be helpful both to other researchers beginning a honeynet experiment of their own and to the developers at the Honeynet project who are constantly working to improve the system.

5 Acknowledgements

Thanks to Ken Patton for getting the Honeywall installation started and for being an essential team member throughout this project. Thanks to our sysadmin Jeff Knerr for tolerating us, giving us equipment, feeding us candy, and providing considerable knowledge and assistance. Thanks to the CS97 class for all the help and moral support. Most of all, thanks to Professor Ben Kuperman for inspiring us to try this project and for providing invaluable help every step of the way.

References

- [1] Bill McCarty. The honeynet arms race. *IEEE Security and Privacy*, pages 79–82, December 2003.
- [2] "Fabien Pouget and Thorsten Holz". A pointillist approach for comparing honeypots, 2005. URL www.honeynet.org/papers/individual/DIMVA2005_Pouget_Holz.pdf.
- [3] Bruce Schneier. The real story of the rogue rootkit, November 2005. URL <http://www.wired.com/news/privacy/0,1848,69601,00.html>.
- [4] Sebek Homepage. URL <http://www.honeynet.org/tools/sebek/>.
- [5] Jaime Sotelo. Get an error with walleye and fixed it, 2005. URL <http://www.securityfocus.com/archive/119/418383/30/0/threaded>.
- [6] Lance Spitzner. The Honeynet Project: Trapping the Hackers. *IEEE Security & Privacy*, pages 15–23, April 2003.
- [7] "The Honeynet Project". URL <http://www.honeynet.org/papers/sebek.pdf>.
- [8] The Honeynet Project, May 2005. URL <http://www.honeynet.org/papers/honeynet>.
- [9] The Honeynet Project. Honeynet project overview, 2005. URL http://honeynet.org/speaking/honeynet_project-3.0.1.ppt.zip.
- [10] Nicholas Wever, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. *WORM*, 2003.
- [11] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: Global characteristics and prevalence. *SIGMETRICS*, 2003.

Profiling HoneyNet Attackers

Connie Li

Taufik Parsioan

December 12, 2005

Abstract

This paper describes the value and process of using honeynets to profile attackers in the blackhat community. Data were taken from a compromised honeynet deployed by The HoneyNet Project and were analyzed for significant events. The data were then used to create a profile of the type of attackers attempting to break in to systems similar to the honeynet and the exploits that may have been used. After an extensive analysis of the alerts given out by the Apache and Snort logs, we find that only inexperienced hackers attack the honeynet for the pure opportunity of it.

1 Introduction

Honeynets are networks of systems that are deployed for the purpose of luring hackers into a network that can monitor their activities. There are two different types of honeynets: production and research. The former are low interaction networks that emulate services for the purpose of protecting organizations while the latter are high interaction networks that provide real services to the attacker [Spib]. Honeynets are valuable because of their ability to collect large amounts of information about attackers and the types of attacks without doing any harm to the system or putting any valuable information at risk. Data collected on a honeynet is also easier to analyze since any activity recorded is assumed to be malicious, given that the network serves no practical purpose.

Honeynet forensics is a specific type of com-

puter forensics in which specialized data analysis techniques are applied only to the data collected on the honeynet. The data provided by a honeynet is almost certainly easier to view than forensic data from any normal system, given that specialized software capturing all activity on a system is often previously installed on honeynets. The goal of honeynet forensics is to recreate attacks from the information collected to obtain a better understanding of what took place after the system has been compromised [RBBK04]. The final results should be able to describe the basic who, what, where, when, and why of an attack.

A subfield of honeynet forensics is profiling using honeynets. Using the results obtained from analyzing the honeynet data, profiling attempts to identify the person or group responsible for the attack and their motives. Profiling is a useful tool because past profiles might help predict what future and current cases of attackers may be like. Profiles of computer hackers also give valuable insight into the blackhat community (hackers working towards negative goals) which is often seen as a mysterious or underground subculture [KAS04].

In the next section, we will discuss in more detail the previous research that has been conducted in the area of honeynet forensics and profiling. We will then go on to discuss our attempt at profiling attackers using data taken from a compromised honeynet. First, we will explain the origin of the data that were used during the project and detail the significant events that occurred on the honeynet. Following that will be our analysis on the data that we've collected. We

will then conclude the paper with our thoughts on how the information we gathered can be used to profile the attackers.

2 Related Work

The field of honeynet forensics, or more specifically, profiling using honeynet data, is a relatively young field of computer security. While there are numerous papers giving step by step analyses of various attacks on regular systems [Che92, Spia], many do not specifically attempt to create a profile of the attacker or identify him or her. Past research mainly attempts to describe how to perform the information gathering stages of computer forensics. There has also been a lot work in establishing stages in the process of honeynet profiling [RBBK04], creating an attacker profile, and finally, being able to classify both an attacker and the exploits used.

Two separate groups of data are left behind by a blackhat after an attack on the honeynet: network clues and system and file information [RBBK04]. The network activity information contains all traffic going to and from the honeypot while the compromised host provides the system logs and tools which were used by the intruder. System information can also be obtained by examining scripts and binaries of rootkits or other files installed or left behind by the blackhat. These two groups of information can be used to create two separate timelines of the events of an attack, which can then be merged and used to answer questions such as who was responsible for the attack [RBBK04].

Once all of the data has been collected, an attacker profile can be created from what is known. An attacker profile is made up of four things: characteristics of the event, consequences of the event, characteristics of the blackhat and characteristics of the target [KAS04]. Characteristics of the event describe why the blackhat might have carried out the attack. These characteristics, such as revenge, greed, or anger, help give a better understanding of the blackhat's motiva-

tion. Consequences of the event describe why the attacker might have chosen the particular target and timing of his or her attack. This helps to determine whether the attack was targeted or merely a random exploit of a known vulnerability. Characteristics of the blackhat discuss the person or group of people responsible for the attack with information such as the motivation, skill, experience, knowledge, nationality, and funding of the attacker. Finally, characteristics of the target give information about the system that was compromised. A set of answers to these questions will help build a profile that may assist in identifying attackers and anticipating or eliminating targets [KAS04].

There have been several attempts to try to understand and define the blackhat community. Hacker taxonomies are often constructed using one or a combination of the following factors: activities, knowledge, motivation, experience and intent. As an example, Kilger, Arkin and Stutzman [KAS04] borrow from the FBI's MICE (money, ideology, compromise, ego) classification of individuals who commit espionage, to create a classification of blackhats based purely on motives. Their six categories, money, entertainment, ego, cause, entrance to social group, and status (MEECES), also appear in other taxonomies such as Chantler's [Cha96], in which there are three groups, elite, neophytes and losers and lamers, which are defined by a hacker's activities, skill level, knowledge and motivation. Chantler went even further to conclude that of the blackhat community, 30% fell into his elite group, 60% were neophytes, and 10% were losers and lamers. Another taxonomy which builds on previous research was created by Rogers, in which hackers were divided into seven distinct, although not necessarily mutually exclusive, categories based on ability: newbies, cyber-punks, internals, coders, old guard hackers, professional criminals, and cyberterrorists [Rog00]. Taxonomies such as these and an understanding of the blackhat community are essential to profiling and identifying specific hackers or hacker groups.

3 Analysis of Log Files

The honeynet logs which we attempted to analyze were taken from a data set provided by The Honeynet Project. The Honeynet Project is a non-profit organization dedicated to improving computer security by providing information about types of attacks, attackers, and motives. They obtain data from various honeynets deployed by members of the Honeynet Research Alliance. This particular data set was published on The Honeynet Project's website for a Scan of the Month Challenge. The Honeynet Project organizes these monthly challenges so that members of the security community can have the opportunity to examine actual honeynet data and share their methods and findings. To reduce the task of extensively searching through the entire set of log files, we initially read through the results of the challenge to learn what significant events occurred on the honeynet.

In total, four different types of log data were provided: Apache logs, Snort NIDS logs, Linux syslogs, and iptables firewall logs. Each data set has a slightly different starting and ending date, but in general, the data ranges from January 20, 2005 through March 17, 2005. Since the Apache and Snort logs will suffice for the purpose of this paper, discussion of the other two logs will be omitted.

3.1 Apache Logs

The Apache logs contain a record of all user activities and errors on the honeynet. They are separated into requests which produced error messages and requests which were successfully processed by the server. Apache logs contain the IP address of the remote system, the time of request, and also the specific request of the attacker.

The Apache logs of this honeynet reveal that the honeynet was compromised using an AWStats.pl exploit on February 26, 2005. AWStats is a server logfile analyzer that graphically generates all web, mail, or ftp statistics. It can also

be run as a CGI in which the program is stored and executed on the web server when requested by a client. In versions 5.7–6.2 of AWStats, the awstats.pl script contained a bug in which a command prefixed and postfixed with the character '—' can be executed on the system. So, if AWStats exists in the cgi-bin directory, running a command such as the one recorded at 21:13:25 on 26/Feb/2005:

- ‘GET/cgi-bin/awstats.pl?configdir=%7cecho%20%3becho%20b_exp%3buname%20%2da%3bw%3becho%20e_exp%3b%2500HTTP/1.1’

will cause configdir to execute the command:

- ‘echo; echo b_exp; uname -a; w; echo e_exp’

which gives attacker information about the system and also reveals user information such as who is logged on the system and what they are doing. This AWStats exploit appears to be relatively simple to execute for a programmer of even little experience. Regardless, the attacker can use it to gain valuable system information by running commands that would go possibly undetected on any regular system given that they would appear to be harmless AWStats commands.

From our data we can see that the attacker uses this exploit to download a tar file twice in the span of a minute with two different IP addresses, once in Italy and another in Germany (shownq below). Given the specificity of the download and the period of time, it seems safe for us to assume that this is the same attacker. From the IP address, we see the attacker downloaded the tar file from a Romanian website shady.go.ro.

- 213.135.2.227-- [26/Feb/2005:14:13:38-0500] ‘GET/cgi-bin/awstats.pl?configdir=%20%7c%20cd%20%2ftmp%3bwget%20www.shady.go.ro%2faw.tgz%3b%20tar%20zxf%20aw.tgz%3b%20rm%20-f%20aw.tgz%3b%20cd%20

```
aw%3b%20.%2ffinetd%20%7c%20HTTP/
1.1''200410'-''Mozilla/4.
0(compatible;MSIE6.0;WindowsNT5.
1;SV1;FunWebProducts)''
```

- 82.55.78.243-- [26/Feb/2005:14:14:43-0500] 'GET/cgi-bin/awstats.pl?configdir=%20%7c%20cd%20%2ftmp%3bwget%20www.shady.go.ro%2faw.tgz%3b%20tar%20zxf%20aw.tgz%3b%20rm%20-f%20aw.tgz%3b%20cd%20aw%3b%20.%2ffinetd%20%7c%20HTTP/1.1''200410'-''Mozilla/4.0(compatible;MSIE6.0;WindowsNT5.1;SV1;FunWebProducts)''

Although we have no way of verifying this, the results of this challenge stated that with this line of code, an IRC bot was downloaded and installed. This seems to be a reasonable claim given that IRC activity was recorded on the honeynet the same day as this download. We also observed that another tar file was downloaded using the same technique and website on March 2. The respondents to this challenge also analyzed this file and found it to be a backdoor to port 60666.

Something interesting that occurred on the honeynet was that on March 12, an attacker (possibly the original) attempted to download the same two tar files from the exact same website. This time however, the request failed, indicating that the version of AWStats on the honeynet was updated to a patched version. There appears to be two reasons for this; either the original attacker was attempting to remove any traces of himself on the system, or another attacker found this machine and wanted to close any vulnerabilities to the system in order to 'own' the machine.

3.2 Snort NIDS Logs

Snort is a knowledge-based, rule-driven intrusion detection system aimed at monitoring system use and detecting any malicious network traffic

or activities. Knowledge-based intrusion detection systems contain information about known attacks and system vulnerabilities and search through system logs for evidence of attacks which are similar in pattern. Intrusion detection systems can also be behavior based, in which information about normal user behavior is given, and any deviations from that behavior is flagged as an attack. The information acquired from a knowledge-based intrusion detection system is usually more accurate but also less complete than behavior-based systems [DDW].

The snort logs recorded from this honeynet give an idea of the types of attacks and probes that any ordinary computer connected to a network is likely to be repeatedly subject to. In total, 85 unique snort alerts were recorded over the period of February 25 through March 31. We will examine and describe a few common alerts.

3.2.1 RPC Alerts

- RPC portmap status request UDP [Classification: Decode of an RPC Query] [Priority: 2]
- RPC portmap listing TCP 111 [Classification: Decode of an RPC Query] [Priority: 2]
- RPC STATD UDP stat mon_name format string exploit attempt [Classification: Attempted Administrator Privilege Gain] [Priority: 1]

If successful, these scans can reveal to any potential attacker the services that are available on the victim hosts. The first alert requests port information for the status service. If this request is successful, the attacker might then attempt to access this service and gain more information about the system.

RPC Portmapper is a server which assigns port numbers to services and is commonly on port 111. The second RPC snort alert appears to be a request to gain information about the services available that were assigned by the

portmapper. While it might be possible that this inquiry is not malicious, an argument could be made that not everyone should be able to access this information or people who need to know this information shouldn't need to inquire for it. Thus, it is reasonable to flag these portmap queries as signs that an attack is about to happen.

The third RPC alert is an attempt to exploit an old string format vulnerability in the rpc.statd service which is sometimes packaged with Linux distributions. The rpc.statd service passes a format string supplied by the user to the syslog() function. The vulnerability in this program was that it neglected to validate the input so that a user could construct a string that would inject machine or executable code into a process address space, which would execute with the privileges of the rpc.statd process, usually root. With these privileges, a malicious user could create or delete any file with the same ease as a root user. This vulnerability in rpc.statd was first noted in 1996 and exploits of it were seen in 2000. The bug has since been fixed and only unpatched RedHat versions 6.2 or older are affected.

3.2.2 MS-SQL Alerts

- MS-SQL Worm propagation attempt [Classification: Misc Attack] [Priority: 2]
- MS-SQL Worm propagation attempt OUTBOUND [Classification: Misc Attack] [Priority: 2]
- MS-SQL version overflow attempt [Classification: Misc activity] [Priority: 3]

The MS-SQL Worm, also known as the Slammer worm, exploits a vulnerability on a Microsoft SQL server, a database management system. It is known as the first Warhol worm, given its capability to infect the entire internet within 15 minutes [SPW02]. In January 2003, the Slammer worm was able to infect more than 90% of computers within 10 minutes and

caused denial of service on several Internet hosts. Systems running vulnerable versions of the Microsoft SQL server were susceptible to heap or stack overflows. Once a UDP packet sent to port 1434 successfully infects a host, its code is executed following either a heap or stack overflow. The code randomly generated other IP addresses and targeted them searching for the same vulnerability. Systems not running Microsoft SQL server, or patched versions of this system can not be harmed by this worm propagation attempt. The OUTBOUND alert informs an administrator that there is an infected machine on the system that is sending out the corrupted UDP packets. This indicates that an MS system is on the honeynet but we don't have enough evidence to verify that. The overflow attempt alert signifies that the UDP packet is trying to execute its code and cause a heap or stack overflow. It makes sense then that the first and third MS-SQL alerts are often seen together.

3.2.3 ICMP PING Alerts

- ICMP PING CyberKit 2.2 Windows [Classification: Misc activity] [Priority: 3]

PING is a network tool that sends packets to a particular host to determine whether or not it is reachable and correctly functioning. It can also report how long it took for the packets to get to the host and back and how many packets were dropped. An attacker can send the ICMP echo request packets and listen for a response to determine whether this machine is active and can be compromised. One of the actions of the W32.Welchia.Worm, seen in August 2003, was to PING the IP address it randomly generated to see if the machine was active and able to be infected.

3.2.4 ICMP Destination Alerts

- ICMP Destination Unreachable Port Unreachable [Classification: Misc activity] [Priority: 3]

The system returns “ICMP Destination Unreachable Port Unreachable” alerts when a packet fails to reach its destination. This can happen if the packet is being sent to a port that is currently closed, or not in a listening state, but it can also happen if the gateway finds a shorter route to send the traffic through. Another possibility for receiving this message is that the gateway does not have enough buffering capacity to forward the packet. Because of the fact that this message can appear in multiple ways, a single alert of this kind does not indicate malicious activity. It must be examined with the other kinds of alerts to see if someone is trying to get access to a port that they are not allowed to.

4 Profiling

4.1 Characteristics of the Target

Knowing the characteristics of our target may be significantly helpful when investigating future attacks, since similar systems are likely to be the next targets of the blackhats who attempted to hack into this particular network. From the honeynet logs provided, we can guess that there were three machines on the system: combo, bridge and bastion. Both the names of the machines on the honeynet and the IP addresses (11.11.*.*) were sanitized by the Honeynet Project. Given that the system deployed is a honeynet, we also believe that it is safe to assume that there was no valuable information (actual or spurious) stored on any of the systems to excessively attract any attackers. It also seems to be a reasonable assumption that there was a relatively low level of security on the honeynet, nothing that would openly try to prevent anyone from attacking the system or try to stop someone once they had compromised the honeynet.

From the snort alerts, it seems reasonable to conclude that many of the attacks or scans attempted were not specific to certain characteristics and services of the honeynet. Commands that were run appear to be relatively simple and

easily repeatable across many systems. Thus, the system was probably unaffected by a majority of these scans and worms simply because they were not applicable to the files and services available on the honeynet.

4.2 Characteristics of the Events

The characteristics of an attack might give us insight into the motives of an attacker. They will tell us what caused the attack to take place. As previously stated, a blackhat may try to attempt an attack to gain revenge, status, information, money, or might try a hack simply for the challenge. None of the attempted hacks on the honeynet seemed to be for economic or political reasons, especially given that fact that as a honeynet, the network likely contained little to no information of value to blackhats with these motives.

4.3 Consequences of the Events

Understanding what the consequences of each event are allows us to understand why a blackhat might have chosen this particular time and target to attack. The results of an attack are often beneficial to the attacker and his cause but can also be harmful if he or she is not skillful enough. The Apache logs indicated that many requests involved gaining root access to the honeynet and/or executing commands to learn more about the system. Given that we are fairly certain that no information of value was on the system, it seems reasonable to state that the main consequence of many of the events is to gain control of bandwidth or more systems to carry out further attacks or propagate harmful worms.

A more specific consequence of the AWStats exploit was the ability to install an IRC bot on the compromised machine. IRC bots typically need to be run on systems with long uptimes and a fast and stable connection to the internet. Thus, there are several advantages if a blackhat manages to find a system that is not his own to run the IRC bot.

4.4 Characteristics of the Blackhats

Given that there were often several events taking place on the honeynet simultaneously, it is difficult to pinpoint one attacker in particular and conclude which events he or she is responsible for. Thus, we will discuss generally the types of blackhats who attempted to break into the system and what their motives might be.

From what we saw in the data, even when we could guess that multiple actions were likely performed by the same blackhat, different IP addresses were logged, indicating that the attacker likely had multiple systems under his or her control. So we conclude that, although helpful, IP addresses are not likely to be conclusive regarding the nationality or location of our attackers. While we also choose not to rely on the times of attacks because of the numerous attackers, we can consider the duration of attacks to determine the amount of resources necessary to carry them out. Resources can be viewed in terms of time and money. None of the attacks attempted required any type of funding other than needing an actual machine to connect to the target. Although it did appear that some attackers used multiple machines to carry out their attacks, they were not ultimately necessary for success. They merely aided in the anonymity of the attacker. The attacks made on the honeynet also did not seem to require much time and dedication. We did not have evidence of any attackers spending an extended, continuous amount of time attacking the honeynet or an attacker consistently returning to the honeynet.

5 Conclusions and Future Work

The profile we created from our data shows that most attacks on the honeynet were done by neophytes, hackers with a basic level of knowledge and experience, but still learning. We came to this conclusion after finding that most attacks on the honeynet were unoriginal, older exploits

for which most systems are no longer vulnerable to. There was also no evidence to support that the honeynet was specifically targeted as acts of vengeance or greed. Most events on the system were simply acts of network or application reconnaissance to find services or vulnerabilities.

It appears that the honeynet provided only basic services and had a limited amount of information, if any. Thus, we conclude that it is unlikely for systems similar to this one to attract hackers above the neophyte level. Although this information is valuable, it is also important to obtain information about attackers of all levels, including the elite level. It is clear though, that elite attackers are unlikely to attack basic honeynets that have no additional means of attracting blackhats. Additional work complementing this project might include deploying honeynets which would attract elite attackers in order to obtain a more complete database of knowledge of the hacker community.

References

- [Cha96] N. Chantler. *Profile of a Computer Hacker*. Infowar, 1996.
- [Che92] B. Cheswick. An evening with Berferd, in which a hacker is lured, endured, and studied. Proceedings of the Usenix Winter '92 Conference, 1992.
- [DDW] Herve Debar, Marc Dacier, and Andreas Wespi. *Towards a taxonomy of intrusion-detection systems*.
- [KAS04] Max Kilger, Ofir Arkin, and Jeff Stutzman. *Know Your Enemy*. 2nd edition, 2004.
- [RBBK04] Frederic Raynal, Yann Berthier, Philippe Biondi, and Danielle Kaminsky. Honeypot forensics. Proceedings of the 2004 IEEE Information Assurance Workshop, 2004.

- [Rog00] Marc Rogers. A New Hacker Taxonomy. 2000.
- [Spia] Lance Spitzner. Know Your Enemy: A Forensic Analysis.
- [Spib] Lance Spitzner. The HoneyNet Project: Trapping the Hackers.
- [SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in Your Spare Time. Proceedings of the 11th Usenix Security Symposim, August 2002.

Building a Neural Network for Misuse Detection

Alan McAvinney

Ben Turner

December 16, 2005

Abstract

One of the fastest-growing areas of computer science research is in the area of security, specifically in intrusion detection. Much research has been done to attempt a functional and useful intrusion detection system, but so far no satisfactory solution has emerged. Recently, attention has focused on using artificial intelligence to train an intrusion detection system (IDS), rather than trying to build one from scratch. A machine learning intrusion detection system has many potential advantages over both human-engineered rule-based expert systems and purely probabilistic approaches. Of the many types of machine learning systems, neural networks offer one of the most promising methods for creating intrusion detection systems that are accurate and manageable. In this paper we present our intrusion detection system, which uses the audit logs generated by the `Audlib` software to train a simple recurrent network to flag system events as either part of an attack, or not part of an attack.

Keywords: intrusion detection, misuse detection, machine learning, neural networks.

1 Introduction

1.1 Intrusion Detection Systems

An intrusion detection system (IDS) attempts to identify the occurrence of unusual, illegal, and/or undesired accesses to a computer or network of computers. The IDS creates alerts which can direct a system administrator or security

professional (which might theoretically be another computer program) to records of attacks, so that data can be recovered, security improved, and, potentially, legal action taken against the intruder. Thus it is important that the IDS have both a high rate of *true positives* (that is, few attacks go undetected), and a low rate of *false positives* (that is, few non-attacks are mistakenly labeled as attacks).

A great deal of work has been done in analyzing the performance of various kinds of IDSs; see Section 2 for a detailed discussion. After a review of the literature, we believe that one of the most fruitful approaches for IDSs in the near future will be in machine learning systems. In this spirit, we implement a neural-network-based IDS. A neural network, specifically a type of neural network called an Elman Network or a Simple Recurrent Network [Elm90], is a learning system which is well suited to tasks which require some temporal or contextual knowledge to complete¹. Our project was to train a neural network to accurately classify system events as attacks or not attacks.

There are essentially two components to any IDS: a record of potentially relevant events on the computer under attack, and an indication of which events correspond to attacks. Most IDS research focuses on the latter, and rightly so; but it is worth noting that the IDS can only be as good as the data it is fed. (For a discussion of how this is an especially important concern in a neural-network-driven IDS, see Section 1.3.) For our system, we chose to use the `Audlib` audit

¹For instance, a neural network can learn to predict that the next bit in the series 01010101 is probably 0.

log generating package [Kup04]. `Audlib` replaces standard library calls with new versions which record the calling process, the arguments to the library call, and other useful information before letting the intended library call go through normally. We then pre-processed the data by parsing `Audlib`'s log files into lists of real values to be fed directly to the neural network as input.

1.2 Theory of Neural Networks

A neural network is based on the idea, inspired by biological neuron-based brains, that many simple nodes, densely connected, can produce complex output. Each node takes a set of real-valued inputs and outputs a single real-valued output; nodes can take their inputs from raw data or from other nodes, and their output can likewise be fed to other nodes or can represent a final result. The weights (rules by which nodes change their inputs into outputs) can gradually be updated through a process known as back-propagation, which trains the network's output towards some specified ideal.

A neural network's nodes are organized into layers, in which each node in layer n is connected to each node in layer $n+1$. When the concept of a context layer is introduced—that is, when part of the network's input on each time step is the values of its hidden-layer nodes from the previous time step—the network gains the ability to 'remember' what its weights were in the past, and the computational power of the network becomes truly impressive—good enough to learn a variety of tasks that would be extremely difficult, if not impossible, to engineer by hand.

It is our belief that among the tasks that can be learned by a neural network is that of deciding when an attack is taking place against a computer system. By feeding the network input that represents some important features of a particular event on the system, we can train the network to recognize events which are likely part of an attack. Furthermore, thanks to its contextual memory, the network can take the events seen previously into account when mak-

ing its classification. The primary task for us as researchers, then, is to properly select features from the thousands of metrics for determining what is happening on a computer system, and present them to the network in such a way that it can make meaningful abstractions and learn the characteristics that define an attack.

1.3 Pros and Cons of Neural Networks

A neural network has several advantageous characteristics that make it an attractive choice for the intrusion detection problem. It is highly robust—resistant to the noise which will inevitably crop up in any real dataset. It is also fast enough (once its training is complete) to conceivably run in real-time on top of a real computer system. Its outputs are not limited to a simple "yes" or "no": they can be probabilistic, and they can be used to sort inputs into arbitrarily many separate categories. Most importantly, perhaps, a neural network has the ability to detect novel attacks, that is, attacks it was never been exposed to during its training. Because it works by creating and refining abstractions from raw data, a neural network learns not just what is an attack and what is not, but what makes an attack an attack.

The neural network approach does have a few significant disadvantages. First, its ability to learn a task is entirely dependent on the input data; as the saying goes: garbage in, garbage out. But this problem is made far worse by the notorious "black box" nature of neural networks: because it is difficult for a human analyst to explain the neural network's behavior in logical terms, it is possible that the network is not learning the same problem that its users think it is.² This black box problem can only be over-

²For example, if the intent is to learn to distinguish nouns from verbs in English sentences, but all the nouns in the training data happen to start with an 'S,' then the network is likely to learn to distinguish words that start with 'S' from other words—and perform very poorly when exposed to new data.

come through careful feature selection and rigorous training on a wide variety of data.

On the whole, we believe a neural network to be well-suited to the intrusion-detection task. The key benefit of a neural network over other types of IDSs, even other machine-learning-based IDSs, is its ability to abstract away from its particular inputs to learn their general characteristics and thus correctly classify new inputs that it has never seen before. In the world of computer security, when new attacks—which are, crucially, not much different from old attacks except in particulars—constantly arise, the ability of a neural network to learn to solve this type of problem is enormously appealing.

2 Previous Research

Intrusion detection systems have in recent years been divided into two groups: those which perform anomaly detection—also called behavior-based systems and those which perform misuse detection—also called knowledge-based systems [DDW99]. Misuse detection systems attempt to determine whether an attack has occurred by scanning the system’s audit data for occurrences of known attacks. Anomaly detection systems rely instead on a statistical analysis of the system’s behavior, signaling out anomalous activity as likely attacks. Each approach has its drawbacks, the most important being that misuse detection systems cannot detect intrusions which do not match the profiles in their set of known attacks, while anomaly detection systems can potentially erroneously classify acceptable behavior as an attack, or conversely may mistake attacks for legitimate use. Machine learning approaches have been suggested for both types of intrusion detection system.

In 1997 Lane and Bradley attempted to build a learning system for anomaly detection [LB97]. Their objective was to learn a profile for each legitimate user in the system, and then examine future actions by the users, using the learned profile to determine if an anomaly had occurred.

User profiles were comprised of sequences of actions (information about command names, behavioral switches, and number of other arguments for each command each user entered at the shell prompt). Once the profiles were built, each subsequent sequence was compared against the appropriate user’s profile; a similarity function determined if the new sequences were normal or abnormal. While the authors may be right to believe their system shows promise, their results are based on data from only four users. It is also worth noting that the system is not truly a learning system, as it makes no attempt to generalize from its input—it simply compares sequences it has seen to new sequences, in a deterministic way that is defined by the authors, not the system itself.

Ghosh, Schwarzbard, and Schatz’s 1999 paper examines three anomaly detection systems, the last of which is a true learning system which employs a neural network to detect anomalies [GSS99]. The authors collected audit data from their network using Sun Microsystem’s Basic Security Module, a built-in auditing tool on Solaris machines. They then tested three techniques on their ability to correctly identify anomalous data: an equality matching algorithm, a simple feed-forward backpropagation neural network, and a simple recurrent network (Elman net). Each system performed better than the one before, and the authors claim that the Elman net could detect 77.3% of intrusions with no false positives, and 100% of intrusions with “significantly fewer false positives than either of the other two systems.” This result demonstrates the potential of a neural network solution to the intrusion detection problem and encourages further research into the performance of Elman nets at the intrusion detection task.

Another application of neural networks is presented in Cannady’s 1998 paper [Can98]. The author mentions the primary shortcoming of rule-based systems for misuse detection, namely that their set of known attacks is extremely unlikely to be complete, and proposes a misuse de-

tection system which utilizes a neural network to learn what characteristics are present in an attack, and then flag future events as attacks or not. The system was trained on a set of data representing network packets, some of which were known to be legitimate and some of which were known to be attacks. When it was presented with another set of similar data that had not been available to it during training, the network correctly identified packets as attacks or not. This result suggests that not only can a neural network learn to identify anomalous behavior, it can also learn the characteristics that are shared by various attacks and apply that knowledge to detect system misuse.

3 Our Experiment

3.1 The Network

To implement our neural network, we used the tools provided by `Pyrobot`, a Python library [BKea]. We set up a standard simple recurrent network with three fully connected layers: input, hidden, and output. (See Figure ??.) The input layer consisted of three components: the audit data, a standard contextual memory layer (i.e. a copy of the previous step’s hidden layer) and a per-process contextual memory layer (a copy of the hidden layer from the last time step in which the data came from the same process it is currently coming from). The hidden layer had N nodes each, and thus each context layer had N nodes also. The output consisted of a single node, which we refer to as “the classification bit” (although in truth it is a real value between 0 and 1, not a binary 0 or 1). The classification bit indicated the presence or absence of an attack; values below a parameterized threshold corresponded to “not an attack,” while values above another threshold meant “attack.”

3.2 Data Gathering and Pre-Processing

The first stage of our experiment involved collection of data using the `Audlib` tool. Two datasets were created: a training set and a testing set. The training set consisted of data that would be used to train the neural network to identify attacks. The testing set consisted of data that would be used to test the performance of the neural network on novel data. Each dataset contained examples of both normal system usage and attacks. To generate attack data, we used a suite of attack tools from www.metasploit.com.

Next we parsed the raw data to process it into a form suitable for use by a neural network. We transformed the data into a set of real-valued inputs. The features available to the neural network were:

- Library Call Name (hashed, normalized to range [0,1])
- For each argument to the system call:
 - argument size
 - argument type
 - argument value (strings are hashed; all values normalized)
- PID of calling process (normalized to range [0,1])
- PID of calling process’ parent process (normalized)
- Real UID of calling process (normalized)
- Effective UID of calling process (normalized)
- Saved UID of calling process (normalized)
- Real GID of calling process (normalized)
- Effective GID of calling process (normalized)
- Saved GID of calling process (normalized)

To avoid inadvertently teaching the neural network to learn that specific UIDs are associated with attacks, we made sure that there was no UID which appeared only in attack data. In the real world, this might not be the case; there might in fact be some user id which was only used by an attacker. But it is far more likely that an attacker would impersonate a legitimate

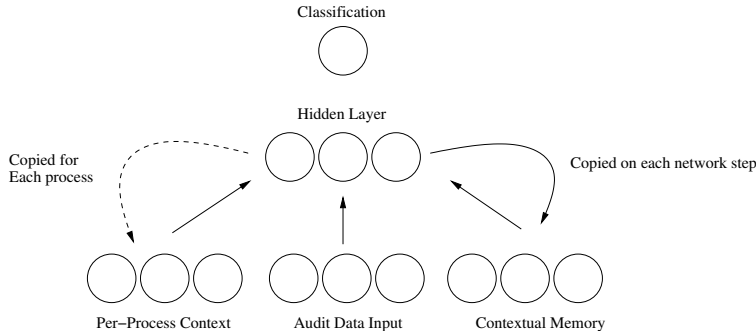


Figure 1: The network architecture.

user (or superuser) and we had to be sure that our IDS could detect this type of activity.

3.3 Training

Once this pre-processing was complete, we could begin training the neural network. Using the timing information from the Audlib tool, we sent input data to the network in chronological order (note that the network itself had no access to the time stamps). The network was trained to output a 1.0 on events known to be an attack and a 0.0 on events known to be not an attack.³ To train the confidence bit, we simply told the network to try for a value of 1.0 when its classification bit matched the expected value, and 0.0 otherwise. The network was trained until its performance on the training data reached an acceptable level of accuracy. To prevent over-training (the phenomenon of the network learning its training data too specifically, thus making it unable to evaluate novel data), we periodically turned off learning and tested the network on the testing data; if its performance on this data was

³There is a small but non-zero chance that some of our events were mislabeled in the training data, because we have no guarantee that an attack was not taking place against the system while we were collecting data. There is no evidence of such an attack taking place, however. We believe that the only attacks present in the data are those that we created ourselves.

better than ever before, we saved the network’s nodes’ weights to a file. Thus the file saves only the weights which performed best on the independent testing data.

4 Results (or lack thereof)

We defined an “epoch” as the amount of time it took to train the network once on each piece of data in the training set. We trained the network for as many epochs as possible before stopping it to evaluate our results. When we did begin to evaluate the results, we found that the network reported a 100% accuracy on its input after the first epoch—that is, the network thought that it perfectly classified each piece of data. Obviously this result is highly unlikely. We are forced to conclude that either our network implementation or our data is flawed, and therefore have no meaningful results to report.

It should be noted that, even had this unexpected bug not occurred, any results from our current data would be preliminary at best. We are confident in the quality of our normal (i.e. non-attack) data, but our attack data, which consists of merely running (unsuccessfully) some standard attack tools downloaded from the Internet, is probably not sufficiently varied to provide a useful basis for the network to learn about attacks in general.

5 Directions for Future Work

This experiment demonstrates nothing conclusively about the viability of neural network-trained misuse detection systems. A more diverse set of attack data is required before a strong conclusion can be made. Thus the first step toward extending this experiment in the future would be a thorough collection of attack data from as many sources as possible. The term “critical mass” is a useful one: a neural network must have enough good input data to learn something meaningful about the general classes of data it encounters, but after a certain point, no new data must be collected because the neural network will have learned everything it can about the problem. So we are hopeful that the classic computer security problem of the hackers remaining “one step ahead” can be overcome with enough effort. Collecting new attack data also has the side benefit of stress-testing the relatively new `Audlib` system. Additionally it would be much better to collect enough attack data to make a significant percentage of the training data attacks, in order to prevent catastrophic forgetting⁴.

An ideal way to collect this attack data would be through use of a Honeypot⁵. This provides a diverse set of attack data with minimal effort from the researcher, as well as data on common types of misuse, and has the benefit of recording the actual actions of hackers, worms, and viruses in the wild, rather than the simulated attacks we used in this experiment.

Once a suitable corpus of input data is gathered, the experiment should be run again in the same manner as described in this paper. Ideally multiple neural networks could be trained simultaneously with varied network parameters, since most of the ideal values for any given parameter

⁴A phenomenon in which a neural network learns a task, then after training on a sufficiently large number of inputs of a different nature, no longer succeeds at the original task

⁵A Honeypot is a computer with no legitimate use—any activity on it must therefore be an attack

of the network (such as the number of hidden nodes, the learning rate, and the momentum, if any) are determined experimentally on a case-by-case basis.

Finally, the viability of a neural network which provides more detailed information about the attacks it identifies should be explored. In the current experiment, the network outputs only one value, giving a “yes/no” answer for the question “Is this library call part of an attack?” But a neural network could theoretically be trained to perform a much more complicated task, such as classifying the inputs into N distinct categories based on what kind of attack they were associated with. An IDS which not only signals that an attack has taken place but also identifies something about the nature of the attack would, one imagines, be quite useful, and the current experiment should be extensible to this problem with relative ease—provided that the input data is carefully classified as it is collected, either by a human expert who knows what type of behavior to expect from a given attack, or by another IDS which has proven to be adept at identifying the most common attacks. Again, the collection of attack data is a time-consuming one, but, thanks to the neural network’s power of abstraction, the task will have a definite (though perhaps not well-defined) end.

6 Acknowledgements

Thanks to Prof. Ben Kuperman for the `Audlib` tool and numerous helpful suggestions and comments; Prof. Lisa Meeden for the `Pyrobot` library and for introducing us to neural networks; the Swarthmore College class of 2006 Computer Science Majors; and everyone who allowed their activity on the system to be logged for our use.

References

- [BKea] Douglas Blank, Deepak Kumar, and et al. `Pyro`: A python-based versatile

programming environment for teaching robotics.

- [Can98] J. Cannady. Artificial neural networks for misuse detection. In *Proceedings of the 1998 National Information Systems Security Conference (NISSC'98) October 5-8 1998. Arlington, VA.*, pages 443–456, 1998.
- [DDW99] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31, 1999.
- [Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.
- [GSS99] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, April 1999.
- [Kup04] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. PhD thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.
- [LB97] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *Proc. 20th NIST-NCSC National Information Systems Security Conference*, pages 366–380, 1997.

Rabbitstew: A Robot Simulator with Variable Morphologies

Ethan G. Jucovy

December 14, 2005

Abstract

I propose to design a realistic physical simulator for robots in Python which can easily use and manipulate automatically generated as well as designed robot morphologies. Following the completion of this simulator I propose to evaluate the relative strength of co-evolutionary methods to conventional evolution in robot bodies using a genetic process to evolve creatures in simulation to perform a simple competitive task. I will allow the physical structures of one population of robots to evolve along with their control mechanisms while a second population remains in a fixed, human-designed body, and the relative successes of these two populations will be compared. The present paper describes my long-term goals and gives the broad implementation details of the proposed simulator.

1 Motivation

In most robotics research, emphasis lies on improving the “brain” or control procedures in a fixed physical form. In evolutionary robotics, control is generally “evolved” (frequently in simulation) through multiple generations by evaluating individual fitness in a large, non-uniform population and applying selection, crossover, and mutation to the population until successful brains are developed. For the most part, this process occurs in a single human-engineered robot body which has proven itself reasonably durable and successful both in and out of simulation, this often

being a model of a commercially available research robot.

In the past fifteen years, however, another trend has been gaining popularity: that of coevolution, or holistic evolution¹, whereby a robot’s morphology, as well as its control parameters, is allowed to vary and be subject to selection pressures.

Several theories have been advanced to explain why holistic evolution is preferable and results in more successful robots. Three major theoretical lines of argument have been advanced. The first is a simple argument from biological observation: brains and bodies evolved together in nature resulting in carefully tuned creatures with brains specifically designed to control the bodies they are in, rather than general purpose brains that can just as easily be used to control completely different bodies. The second, related argument takes a developmental perspective: holistic evolution avoids potentially constraining human bias and, proceeding by a large number of small and gradual modifications to both brain and body, allows a tighter coupling of brain and body to emerge.[7] The third is a theoretical mathematical argument which will be discussed later. However—somewhat surprisingly—no direct, side by side comparison has ever been done to confirm that holistic evolution *is*, in fact, a better approach.

For this reason, I propose to compare holis-

¹The accepted term in the literature is the former, “coevolution”; however, following [9], I prefer the term “holistic evolution” as more descriptive and less ambiguous.

tic evolution to the conventional evolution of control parameters in fixed morphologies². I will evolve two populations of robots in simulation using a competitive selection method to maximize evolutionary pressures: one population will contain robots with varying, initially random morphologies, and will be evolved holistically; the other population will have only its control structures evolved within a fixed architecture modelled on a real, commercial research robot such as an ActivRobots Pioneer (www.activrobots.com) which is generally accepted to have good design and to have the ability to perform well in a wide variety of tasks.

By the nature of the competitive, variable fitness function, which depends crucially on the other members of the population, there is no obvious, objective way to compare the relative success of two populations of robots. As a substitute, at periodic intervals throughout the evolutionary process I will take the best-performing members of each population and compete them against each other to determine which population has evolved “more successfully.” As my sympathies do lie with the proponents of holistic evolution, I expect that, *eventually*, the holistically evolved population will outperform the other, perhaps by quite a consistent and wide margin, though I expect that the opposite will be true during the early stages of the evolutionary process.

In order to perform this experiment I will need a simulator which allows robot morphologies to be specified easily and modified automatically without additional code compilation or intervention from the user. Unfortunately, no such simulators are currently available. Therefore my first step must be to develop a simulator which will allow me to run this experiment.

In the present paper I describe the details of both my proposed simulator, the Robotic

²For lack of a better term, I will refer to this as “conventional” evolution.

Artificial Brain/Body-Intertwined Simulation Toolkit and Evolution Workshop, or Rabbitstew, and of the experiment that I will perform on the simulator. In the first section I discuss related work in the field of holistic evolutionary robotics, and I describe the present state of commercially and freely available robot simulators; in the second section I give the proposed implementation details of both the physical simulator itself and the separate three-dimensional graphical display program; and in the final section I describe the experiment which motivates the design of the simulator.

2 Related Work

2.1 Holistic Evolution Research

Salmon[9] offers a more complete overview of work done on holistic evolution and summarizes some of the empirical and theoretical arguments in favor of holistic evolution.

2.1.1 Proof of Concept

Karl Sims[10, 11] successfully demonstrated holistic evolution, essentially as a proof of concept; starting with completely random morphologies, spreading control structures across the robots’ bodies, and giving the robots a competitive task to complete, he produced a number of interesting robots with widely varying morphologies, many of whose strategies in the competition were essentially quite simple and depended crucially on their particular physical structures in ways which conventionally evolved robots could not have developed in a fixed form.

Sims evolved five groups of creatures, each with a different fitness metric: one group was evolved to quickly take control of a block in the center of an arena and prevent an opponent creature from doing the same; one group was evolved to locomote by swimming in an underwater environment; one group was

evolved to walk on land, another to hop in a low-gravity world, and another to quickly follow a point of light. However, he did not compare this in any way with non-holistically evolved robots as his motivation was primarily to create thought-provoking computer art and to demonstrate the feasibility of holistic evolution rather than its advantages.

Thomas Ray[8], in a follow up to Sims' work, implemented a similar procedure, replacing the automated fitness selection functions of the original study with ratings provided by human observers to select for increased aesthetic and emotional appeal of the resulting reatures.

Somewhat troublingly, Ray estimates that up to 90% of over 300 "interesting genomes" that resulted, many of which were able to swim and crawl successfully, were found by the random generation which initializes a population. However, he does say that evolution improved his creatures, and it is unclear whether the 90% he cites were found *purely* by random initialization or whether some they were enhanced by at least some evolution.

Furthermore, Ray was selecting for subjective aesthetic qualities more than for stability and success at a well-defined task in a physical environment; indeed, one of the creatures he selected for reproduction had an unfortunate tendency to explode under the strain of its own internal forces. It is likely that, with more physically challenging selection pressures, the importance of evolutionary processes would increase while the likelihood of finding successful individuals purely by random search would decrease significantly.

Pablo Funes and Jordan Pollack[6] provide independent supporting evidence for the potential of physical structure evolution in addition to control evolution. To demonstrate that physical structure can effectively be generated by a genetic process, they evolved passive objects such as bridges, cantilevers, and crane arms in simulation using simulated

Lego³ blocks. Perhaps confirming the developmental argument that undirected search techniques such as simulated evolution remove restrictive human biases, their resulting structures were often unusual and counterintuitive, but perfectly functional even when reproduced with physical Lego blocks.

2.1.2 Advantages of Holistic Evolution

Providing some evidence of the value of holistic evolution, Balakrishnan and Honavar[1] performed a semiholistic evolution in a very limited, idealized environment, where fitness measures were fixed based on success at a block-pushing task. In their first trial they performed conventional evolution of neural networks in a fixed structure: a robot with eight non-intersecting short-range sensors. In subsequent trials the number, range and placement of sensors on the robot could be modified during the evolutionary process. Using this limited holistic evolution, performance as measured by the static fitness function did not drop, and, interestingly, the number of sensors was generally minimized despite no pressure for efficiency being built in. In addition, when sensor range was evolved in addition to number and position, peak performance increased while sensor number still remained relatively low. These results suggest that holistic evolution may help robots discover solutions that are both more effective and more efficient than would be discovered with conventional evolution in a fixed form.

Further evidence that holistic evolution can improve efficiency was provided by Bongard and Paul[3], who also used a limited form of holistic evolution to show that performance and efficiency can be improved by allowing evolution to modify physical structure. In this case, variable parameters in the morphology consisted of the lengths of the body segments and the dimensions and placement of weights on the legs of a biped robot whose task was

³Lego is a registered trademark of the Lego group.

to walk as far as possible in a given amount of time. The populations with variable morphologies both performed better on average and had higher peak performances than those with static morphologies, providing additional empirical support for holistic evolution; however, as in Balakrishnan and Honavar’s work, the holistic evolution performed was fairly limited.

Conrad[4], meanwhile, provided a theoretical argument in favor of holistic evolution, called the extradimensional bypass. Any complex fitness landscape will contain peaks and valleys, and two adaptive peaks which may be separated by a valley in a low-dimensional landscape may be connected by an adaptive ridge if additional dimensions are added to the landscape. As a holistic approach to evolution includes more free parameters than a conventional approach, the fitness landscape consists of many additional dimensions; so adaptive ridges may exist between what would be peaks in a lower dimensional (fixed body) manifold of the same space.

Stober and Gold[12] performed a variety of experiments, with tasks ranging from wall following to object gathering, to evaluate the potential advantages of evolving neural network morphologies, as opposed to merely evolving the weights of the network as in conventional evolution. They found, however, that performance did not improve significantly when network morphologies were allowed to evolve, and in fact successful strategies found by the two populations were quite similar.

2.2 Existing Simulators

There exists at present no simulator up to the task of performing holistic evolutionary development. Sims’ work was done on a massively parallel architecture with custom code that cannot be made publically available. Ray, Bongard and Paul used MathEngine, an apparently now defunct real-time physical simulation package produced by MathEngine PLC,

Oxford; Bongard’s more recent research and that of Pollack have used code which employed the Open Dynamics Engine and are not publically available.

The open source Player/Stage package does contain a three dimensional rigid body simulator, Gazebo, designed in ODE and OpenGL, as well as Player itself, a network server for robot control. However, specifying a robot morphology for use in Player and Gazebo is slow and labor intensive; a Player device must be developed to specify how the robot interacts with its sensors and actuators, while a Gazebo model must be hard-coded and compiled to describe the robot’s physical body. Other simulators are equally problematic: Webots and JRoboSim require users to describe robots similarly to the Player/Stage project and are not intended for morphological evolution; the Graphical Workshop for Modelling and Simulating Robot Environments (Gwell) and Robsim only allow robot models to be hand-created by the user in a visual runtime environment; Easybot requires precompiled libraries to specify robot controllers and does not perform collision detection or other physical modelling.

The Laboratory of Intelligent Systems (LIS) at the Ecole Polytechnique Fédérale de Lausanne has been developing several evolutionary robotics tools including Enki, a fast two-dimensional simulator capable of simulating large groups of robots; Teem, a software framework for evolutionary robotics experiments; and Goevo, an application for evolving neural network controllers for real or simulated robots. Unfortunately LIS does not currently have any tools available for evolving robot morphologies, and the pieces of their evolutionary framework that do not directly rule out morphological evolution are not designed with this in mind.

Table 1 contains download and documentation locations for the software mentioned above.

3 Rabbitstew

3.1 Tools

I will write the simulator in Python. For the physics of my simulator I will use PyOde, a set of Python bindings for the Open Dynamics Engine (ODE), a native C library for rigid body dynamics with built in support for fast collision detection and joint connections between bodies. ODE is quickly becoming a standard library for simulation of evolutionary robotics and will be well suited to this task. I will implement a visualization of the simulation separately using the Visual module for Python. Neural network implementation will be carried out using the Conx module of the Pyro programming environment. See Table 1 for locations of online resources for this software.

3.2 Simulator Implementation

Rabbitstew is composed of two major parts: the data structures which fully describe robot genotypes for storage and manipulation, and the physical instantiation of the robots active in the simulation. Passive features in the simulation environment are not considered to be a separate category but are instead treated identically to robots; a passive structure such as a block or a wall can be defined simply by creating a robot with no brain.⁴

3.2.1 Robot Genotypes

The design of robot genotypes in Rabbitstew is broadly based on that described by Sims[11] with elements of Ray's[8] implementation.

An individual robot genotype will be represented by a directed graph which is composed of Nodes and Connections including an arbitrary root Node. Each Node represents

one body unit and contains a Segment and a list of Connections. Each Connection represents a physical attachment between the two body units specified in the parent and child Nodes and will contain a child Node, a relative position, orientation, and scale, two bits representing joint type (hinge, ball, slider or fixed), and a direct-recursive limit parameter since circuits are permitted in the graph. To prevent infinite indirect recursion, robot phenotypes will have a universal, externally imposed, user defined size-ratio limit, a restriction on the number of body segments relative to the size of the genotype.

A Segment will consist of two bits representing shape (box, sphere, or cylinder) and physical dimensions for the Segment, with the number and type of dimensions dependent on the particular shape of the Segment. Shape dimensions are relative and will be normalized to have identical volume; absolute dimensions of a given Segment will be determined by a combination of the parent and child Segments' dimensions and the scale factor of the connection between them.

Each Segment will also contain a Brain which is represented in genotype as a directed graph of neural units with graph connections storing the network weights between units. Three types of neural units will be available: Sensors, which get input from the environment, Effectors, which send torque outputs to the parent Joint of the Segment in which they are located, and Neurons, which are purely processing units with inputs from and outputs to other neural units. Available Sensors will include binary contact Sensors which are active if and only if the associated Segment is currently in contact with any other physical body and sets of three direction Sensors which give the normalized direction from the center of the associated Segment to a particular source (available sources will be the center of the target and the center of the root Seg-

⁴This may be changed in a later release of the simulator if it is found to be unsatisfactory; its major advantage, and the motivation for this organization at least in the initial release, is its simplicity.

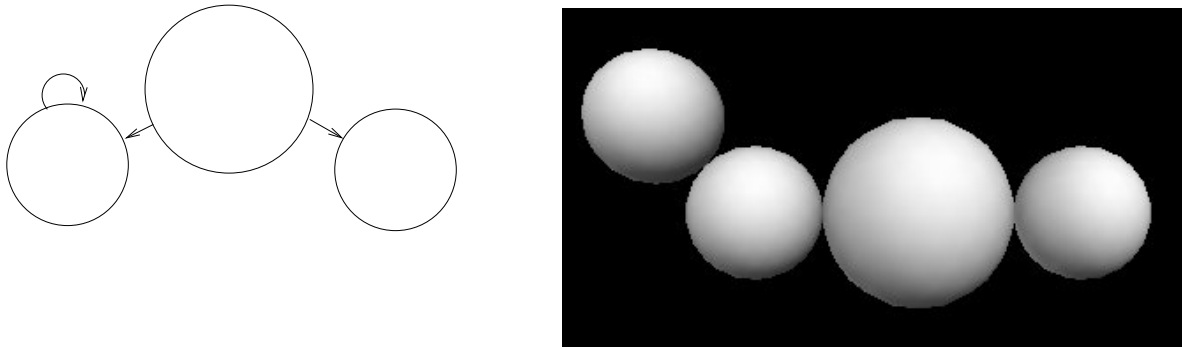


Figure 1: A sample morphological graph (attributes and parameters, including the single-level recursive limit parameter for the circuit, not shown) and the resulting structure.

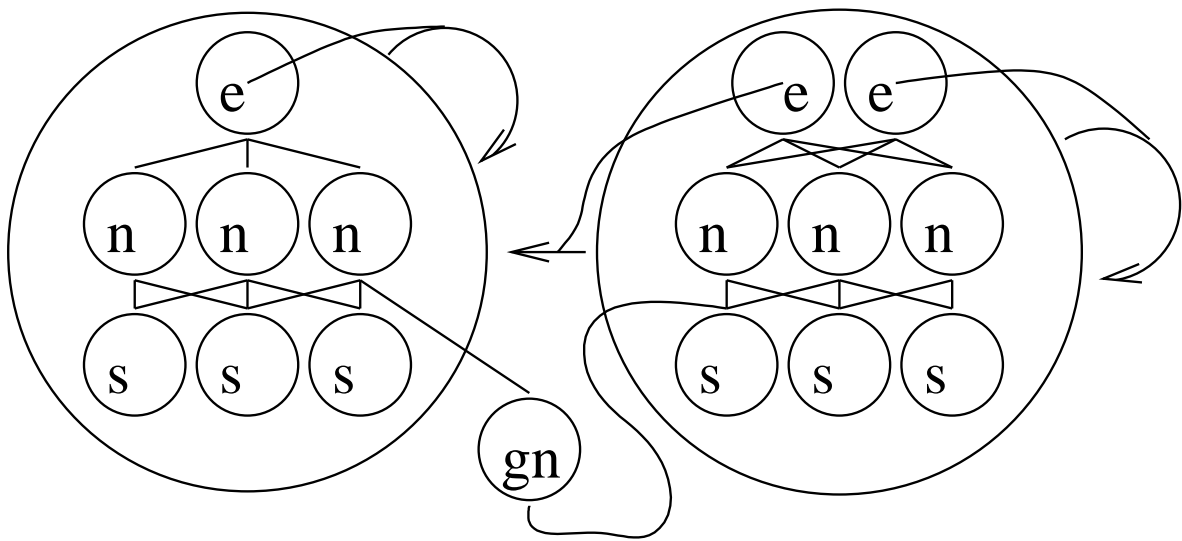


Figure 2: A graph with embedded and global computational units (Neurons, Sensors, Effectors) shown. Effector outputs to local joint connections are represented by lines to the relevant connection.

ment of the opponent in the competition).⁵

A global Brain, not associated with any particular Segment, will also be available; this Brain will consist entirely of Neurons but can be connected to any localized neural units to permit centralized control. With the exception of this central Brain, all individual neural units in a Brain can only be connected to other units in the same Brain.

Note that there are no constraints on the number or types of neural units in a Brain aside from the limitation that the central Brain, if it exists at all, be composed only of Neurons. Therefore it is possible to create a fully distributed robot controller, by using no centralized Neurons at all, or a fully centralized one, by limiting the embedded neural units only to Sensors and Effectors, in addition to hybrid controllers with some localized and some centralized processing.

3.2.2 Physical Structures

When the simulation starts, each robot which will be active in the competition is synthesized from its genotype; when the simulation is terminated the synthesized robots are destroyed. While this can result in a considerable number of redundant graph traversals if a robot is active in multiple simulations, the redundant computation is only done at the initialization of the simulation and therefore should have no effect on individual simulation performance, and the memory saved by “cleaning up” robots between simulations, which would affect simulation performance if too much memory was used, more than offsets the additional time to resynthesize robots.

⁵Additional Sensors and Effectors could be made available later and might result in more interesting and varied behavior, but for simplicity the initial sets of Sensors and Effectors will be limited to these few. Additionally, the direction Sensors should and will ultimately be generalized to give the direction to any arbitrary (fixed or moving) target for a more general-purpose simulator that can be used for other experiments.

Synthesis of a robot proceeds from the designated root Node. In an attempt to increase the number of graph nodes that are actually synthesized into at least one body part before the size-ratio limit halts the process, a breadth-first search will be used to traverse the graph. With this method, however, a low size-ratio limit combined with high recursive limit parameters on connections could still result in creatures with some graph nodes that are not synthesized into any body parts. This is intended to make the evolutionary process more complex and realistic, since many genetic features in nature are passed down without manifesting in every generation.

It should be noted that this approach still results in a direct, one to one mapping between genotype and phenotype; such a direct mapping has been argued[5] to be problematic in evolutionary processes as it ignores the many stages of growth, does not scale well to large organisms, and fails to impose constraints of symmetry, which has been demonstrated to increase organisms’ efficiency[2]. Regardless, I feel that the current model is a reasonable compromise between genotype-phenotype complexity and simulator simplicity.

At each Node up to three ODE objects are created and initialized: a Body, which primarily contains an object’s mass properties; a corresponding Geom object, which describes the object’s spatial extent for collision detection; and, for all Nodes but the root, a Joint between the object’s Body and the Body of the parent Node, which physically links two objects and provides motion constraints between them. Each robot will be stored in a list of Geoms and an ODE JointGroup. The Geom class has a function which returns the Body to which it is linked so no separate list of Bodies needs be maintained.

At each Node the corresponding neural network will also be synthesized from the local graph describing a Brain; following the synthesis of individual units the central Brain will

be constructed as well. Each Brain will be implemented in the physical simulation as a standard neural network.

3.3 Visualizer Implementation

At the start of a physical simulation the specifications of each robot are written out to a data file for subsequent use by the visualizer. Each body unit of each robot is described, in turn, first by a number specifying the shape of the unit and then by that unit’s *absolute* dimensions, which can vary from one (for a sphere) to three (for a box) floating-point numbers.

At each n timesteps during the simulation (by default $n = 1$ but this can be overwritten by the user) the state of the simulation is written to the data file. This consists of a three dimensional position vector and an orientation quaternion (both provided by ODE accessor functions) for each ODE Body, where the states of Bodies are given in the same order as the initial data. A single small robot with three body units, therefore, would yield a state output of 21 single-precision floating-point numbers per output cycle.

When the visualizer starts it will first read in unit data, one unit at a time, creating a three dimensional model of each unit using the Python Visual module and storing the units in a single list. Models will then be initialized to the starting position and orientation given by the first set of state data and all models will subsequently be set visible. Visualization proceeds in a loop by reading in each set of time-dependent state data and updating all models accordingly until all data has been processed or the visualization is terminated by the user.

For spheres and boxes, position in three dimensions is given directly by the first three values of the 7-tuple representing a unit’s state. Unfortunately slightly more computation is necessary for cylinders, as ODE represents a cylinder’s position by its center of mass and the Visual module represents a cylinder’s

position by the central point at one end of the cylinder. Orientation is determined from the last four values which give the unit’s orientation quaternion q by setting the object’s axis vector to the last three terms of the result of the operation $q' \times (0, 1, 0, 0) \times q$ and its up-directional vector to the final three terms of the result of the operation $q' \times (0, 0, 1, 0) \times q$, where $q' = (q_0, -q_1, -q_2, -q_3)$.

4 Long-Term Experiment

After the simulator has been developed, I will run two evolutionary algorithms, one on a population of randomly-generated morphologies and employing holistic evolution; the other on a population with uniform, engineered morphologies and evolving only the weights of the neural network control structures.

I intend to use a all-versus-best two-at-a-time competition⁶ similar to that of Sims[11], where pairs of robots from the population will be physically simulated as they compete in a time-limited zero-sum game. The evaluation of fitness I will use will be a ratio of the two robots’ center-of-mass distances from the center of the “world” after a short period of simulated time; so robots would have to find strategies both for quickly reaching the goal point and for preventing their opponent from doing so. This competitive, zero-sum fitness function should result in fairly fast and dynamic evolution with a wide range in populations.

At periodic intervals throughout the evolutionary process I will select the two or three highest-performing members of each population in the current generation and I will have these “champions” compete, in the same task, with one another. These interpopulation competitions will have no effect on the evolution of

⁶Variations on competition size and structure are of course possible and could be employed for follow-up experiments.

Software	URL
Easybot	http://iwaps1.informatik.htw-dresden.de/Robotics/Easybot/
Enki	http://lis.epfl.ch/resources/enki/
Goevo	http://lis.epfl.ch/resources/evo/
Gwell	http://diablo.ict.pwr.wroc.pl/\%7epjakwert/
JRoboSim	iwaps1.informatik.htw-dresden.de/Robotics/JRoboSim/
MathEngine	www.mathengine.com (website down)
ODE	www.ode.org
Player/Stage	playerstage.sourceforge.net
PyOde	pyode.sourceforge.net
Pyro	www.pyrorobotics.org
Robsim	http://www10.brinkster.com/geniusportal/robsim.html/
Teem	http://lis.epfl.ch/resources/teem/
Visual	www.vpython.org
Webots	www.cyberbotics.com

Table 1: Software locations.

each group; they will be used simply to measure and compare the progress of each population.⁷ In this manner I will evaluate the relative success of each population as compared to the other at different stages of evolution.

I predict that in the early stages of evolution the holistically evolved population will perform far worse than the stable-body population, by virtue of the latter’s having an effective and functional body to work with; the latter group essentially has a significant head start by having a pre-engineered body. However, I predict that this will eventually reverse, and that the holistically evolved population will ultimately outperform the stable-body population, for several reasons.

First, by having their brains and bodies develop simultaneously, the holistically evolved robots will be able to take full advantage of their physical structure and will be able

⁷Some obvious alternatives include periodically pitting the best performer of each population against each of the members of the opposite population at the same generation, or competing each member of the two populations; best vs. best has the (dubious) advantages of simplicity and quicker processing time, but other methods could be used for additional follow-up analysis or for further experiments.

to evolve relatively uncomplicated strategies that hinge on the particular constraints and idiosyncrasies of their bodies, and they will simultaneously be able to modify their physical structures to complement and build upon the strategies that they have already developed.

Second, their population will be considerably more variable and therefore selection pressure to find general-purpose and adaptable strategies may be stronger than in the stable-body population. And third, as posited by Conrad[4], the extra dimensions of the evolutionary search space may allow for additional “ridges” in the solution landscape connecting what in a lower-dimensional space would be two separated peaks.

Following this experiment a number of avenues for further study will be available, involving modifications of a number of parameters in the experiment. In addition to varying the competition structure and the sensors and body parts available to the evolving robots, a variety of tasks could be employed besides the proposed “control the center” goal. Unfortunately it is quite difficult to predict the effects of any of these modifications, so I intend to consider them only after completing

the present experiment.

5 Acknowledgements

Many thanks to Josh Bongard, Douglas Blank, Claudio Mattiussi, Jordan Pollack, John Rieffel, Branen Salmon and Karl Sims for their support and their advice regarding presently available simulators. Jordan Pollack, John Rieffel and Josh Bongard also referred me to the Open Dynamics Engine. As noted above, many of the implementation details of my simulator are due to Karl Sims; much of the organization of the project resulted from recommendations by Branen Salmon. Thanks also to Cortland M. Setlow who explained the method described above for determining visual orientation by quaternion multiplication, and finally to Ben Kuperman for his frequent encouragement and review of my project.

References

- [1] Karthik Balakrishnan and Vasant Honavar. On sensor evolution in robotics. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, 28–31 1996. MIT Press.
- [2] Josh Bongard and Chandana Paul. Investigating morphological symmetry and locomotive efficiency using virtual embodied evolution. In J.-A. Meyer et al., editor, *From Animals to Animats: The Sixth International Conference on the Simulation of Adaptive Behaviour*, 2000.
- [3] Josh C. Bongard and Chandana Paul. Making evolution an offer it can't refuse: Morphology and the extradimensional bypass. *Lecture Notes in Computer Science*, 2159, 2001.
- [4] Michael Conrad. The geometry of evolution. *Biosystems*, 24:61–81, 1990.
- [5] Frank Dellaert and Randall D. Beer. Toward an evolvable model of development for autonomous agent synthesis. In R. Brooks and P. Maes, editors, *Artificial Life IV Proceedings, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, 1994. MIT Press.
- [6] Pablo Funes and Jordan Pollack. Computer evolution of buildable objects. In P. Husbands and I. Harvey, editors, *Fourth European Conference on Artificial Life*, pages 358–67, Cambridge, MA, 1997. MIT Press.
- [7] Jordan B. Pollack, Hod Lipson, Gregory Hornby, and Pablo Funes. Three generations of automatically designed robots. *Artificial Life*, 7:215–23, 2001.
- [8] Thomas S. Ray. Aesthetically evolved virtual pets. In Carlo C. Maley and Eilis Boudreau, editors, *Artificial Life VII: Workshop Proceedings*, Portland, OR, 2000. Reed College.
- [9] Branen Salmon. Embodied evolution in a morphologically heterogeneous population of robots. <http://web.cs.swarthmore.edu/~meeden/cs81/projects/salmon.pdf>, 2003. Swarthmore College Senior Seminar Project.
- [10] Karl Sims. Evolved virtual creatures. In *Computer Graphics Annual Conference Series*, pages 15–22, July 1994.
- [11] Karl Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV Proceedings, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, 1994. MIT Press.

- [12] Jeremy Stober and Jonah Gold. Evolvable morphologies for robot controllers. <http://web.cs.swarthmore.edu/~meeden/cs81/projects/stober-gold.pdf>, 2003. Swarthmore College Senior Seminar Project.

Sweeter Honeynets

Kenneth Patton

December 2005

Abstract

The honeynet is a new technology used in the field of computer security for researching the actions of hackers. While honeynets have the potential to give us great insight into the hacker world, recent studies have shown that the rate of data collection by honeynets is far from optimal. This paper first discusses the motivation for using honeynets to track hacker's activities as well as a brief background on honeynets and the various types of hackers in the world. Then, solutions to the problem of increasing honeynet traffic are presented. The primary solution that the author develops is using a webserver as bait, and the steps needed to implement this approach are discussed in detail. Two additional methods are also presented as alternatives - advertising through hacker chatrooms and online contests. The author concludes that all three of these methods are feasible and intends to implement them for experimental confirmation.

1 Introduction

With computer systems playing a larger role in society today, computer security is now more important than ever to protect the confidentiality, integrity, and availability of digital information. Computer systems are also becoming increasingly complex, making it difficult to insure system security when hundreds of applications are run on a regular basis with just as many running in the background continuously. In addition, application developers typically possess a release and patch mentality in order to minimize the

time to market, increasing the number of software bugs that external computer crackers can use to compromise a host system.

Due to the prevalence of hackers on the internet today, we recognize the need to research the methods that hackers use to compromise target systems. There are many different techniques employed by hackers to compromise external computers; these range from code analysis and the manual design of tools to the less sophisticated downloading of automated scripts from the internet. Ideally we would like to obtain information about all the different types of attacks that hackers employ, although the less sophisticated attacks are generally more prevalent.

In order to track the actions of hackers, we need a strategy for allowing hackers to do their work while they are unknowingly observed. One tool that is used to facilitate this is known as a honeynet. In a honeynet, a single secure computer monitors a group of insecure "bait" computers that are waiting to be compromised by external hackers. While still a relatively young technology, honeynets help facilitate research into computer crimes because they present hackers with an otherwise undisturbed environment, which makes it simple to identify what traffic and actions on a computer in the honeynet are due to hackers.

There are typically two types of honeynets in use today: production honeynets and research honeynets. Production honeynets are simple honeynets used to capture limited amounts of information, often employed by companies to help protect more valuable systems on a net-

work. Research honeynets are more complex entities designed to capture as much information as possible about the behavior of intruders. Research honeynets are typically not designed to protect other systems on the network in the short-run, but ideally benefit systems in the future through analyzing the techniques that hackers use to compromise typical machines. All traffic on research honeynets is known to be intrusive in nature because they have no other intended purpose, which makes it easier to analyze a hacker's behavior.

However, as research honeynets are typically unadvertised, they attract relatively low amounts of traffic. In a recent study [1], the average amount of time it took an unpatched Linux system connected to the internet to become compromised was approximately 3 months. While data collected from individual break-ins is certainly valuable, with such sparse occurrences it is questionable whether this is the best method for collecting data. Instead, by making the honeypots more visible through actively advertising them, we can draw more hacker activity at the cost of additional legitimate traffic. As an example, by placing a webserver on a honeypot and designing a simple but enticing website that draws a small amount of web traffic, we present a bigger target for typical hackers than an anonymous machine on the network. Unfortunately this has the drawback that not all of the traffic on the machine will be illicit, but since we know exactly what traffic to expect it should not be difficult to filter out attacks on the machine. Standard web browsers that request valid pages of the website will not be considered attack traffic, but web requests for invalid pages and non-http traffic will be considered attack traffic.

2 Background

2.1 Hackers

There are a number of different types of hackers, each with different motivations and meth-

ods of attacking a remote computer. We classify as hackers individuals who, through direct or indirect action, causes a machine to behave in a manner other than intended by the owner. Often this results in the hacker gaining control over the system, but we still classify individuals who make the machine behave abnormally but do not gain control of the system as hackers (for example, due to DDoS attacks). Here we try to classify the different types of hackers that might typically be encountered by a system on the internet and their motives. A more in-depth classification of hackers is presented by Marc Rogers in [5].

The Accidental Hacker

An accidental hacker is a user who, without previous intent, unknowingly compromises or disrupts the normal behavior of a supposedly secure computer. The user may realize the result of their actions after the fact but generally will not try to exploit the vulnerability that they found. Obviously such a user has no prior motives, which makes it difficult to attract these accidental hackers. Generally if a system vulnerability can be taken advantage of accidentally, it is a serious threat to the security of the computer and be prone to intentional exploitation by less scrupulous hackers. Since break-ins due to accidental hackers are often due to glaring security holes and occur sporadically, they have little research value when focusing on typical hacker threats.

Worm and Virus Creators

In general virus and worm designers do not directly attempt to compromise particular machines on the internet, but through their actions they indirectly account for a portion of system break-ins. Their motives are often simply entertainment, but occasionally they write viruses for a purpose - usually with motives similar to those of blackhats. However, the automated nature of viruses and

worms cause them to attempt to compromise machines in the same way every time, making their actions very predictable.

”Script Kiddie”

”Script Kiddies” are relatively unskilled hackers that use automated tools downloaded from the internet in order to attempt to break into machines. These are the most prevalent type of intentional hackers, but generally their actions are easy to reproduce and identify. ”Script Kiddie” motives typically range from simply the excitement of doing something illegal to collecting botnets for DDoS attacks and harvesting credit card information.

Blackhat / Cracker

Blackhats, often called crackers, are experienced hackers that are typically motivated to break into a protected computer system for personal benefit such as money or access to sensitive data. As the most advanced type of hacker, they generally have detailed knowledge about system exploits and are able to carefully take advantage of those exploits in order to gain full control of a system. Blackhats motives can range from the thrill obtained due to the challenge of hacking a machine to disgruntled employees trying to get back at an employer.

Whitehat

Whitehats are similar to blackhats, but differ in their motives for breaking into a machine; while blackhats compromise machines for personal benefit, whitehats claim to be ”ethical” and break into machines in order to help make computer systems more secure. The difference between whitehats and blackhats can be narrow at times, with whitehats on occasion breaking into machines in order to investigate blackhats, but generally whitehats will leave less of a trace on a compromised machine than a blackhat would. Whitehats are often employed by

security companies and rarely act on their own to compromise random hosts on the internet.

From a computer security standpoint, the most valuable information we could obtain would be the strategies used by blackhats / whitehats to break into systems, followed by information about typical ”Script Kiddie” exploits and toolkits, and lastly how viruses and worms penetrate machines. Unfortunately, the traffic on an average machine connected to the internet will generally find more break in attempts in the reverse order; worms are likely to generate the most, albeit relatively simple, ”attack” traffic, while ”script kiddies” generally make up a much larger percentage of attempted break ins than blackhats. Regardless, we need a tool to investigate the methods that these different sources employ to compromise machines: a honeynet.

2.2 Honeynets

A honeynet is a collection of computers whose purpose is to track everything that occurs on designated ”bait” computers, the honeypots. Honeypots are not used for any particular function on the network, but rather exist solely to be broken into by external hackers. The goal of a honeynet is to research the actions of hackers, which is best accomplished on honeypots since they contain essentially only attack traffic with little background noise.

In a basic honeynet setup as seen in figure 2.1, all traffic passing between local computers and external computers on the internet must pass through a honeywall. The job of the honeywall is similar to a firewall, with advanced filtering and logging capabilities. The honeywall may be set up in one of two configurations: as a standard network bridge or using Network Address Translation.

For the purposes of strictly research in a non-production environment, the honeywall is best set up as a network bridge because it allows for multiple honeypots and the honeypots to appear

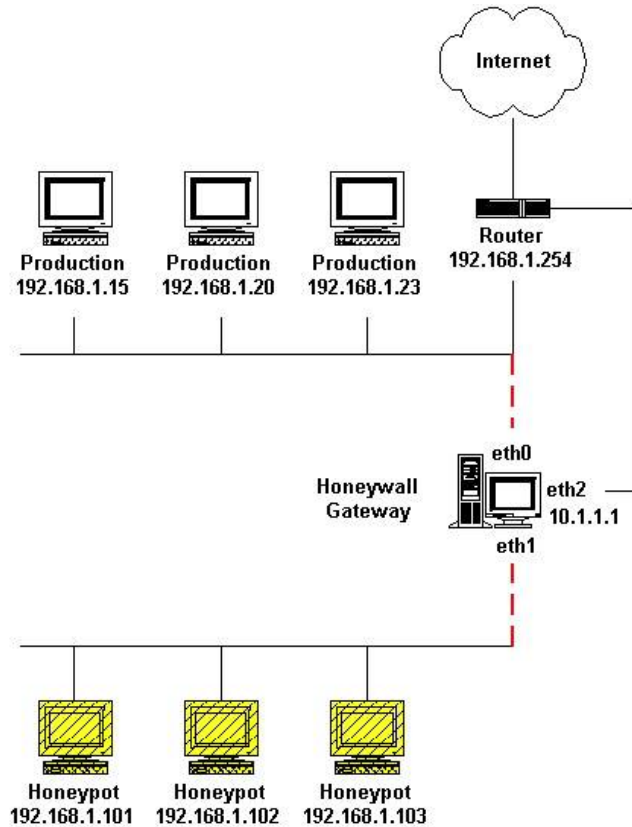


Figure 1: Standard Honeynet Setup

as standard machines on the network to hackers. In an environment where the honeynet is set up as a decoy to distract hackers from other critical network resources, NAT can be useful because it allows the honeypot to camouflage the production machines. For example, a company could put a webserver behind the honeywall and have all communication on port 80 forwarded to it while all other is forwarded to a honeypot. Overall it looks like the honeypot and the webserver are the same machine, and a hacker will be distracted trying to break into the honeypot while the webserver is fully secure. This allows the company to analyze the hacker's actions in addition to quickly highlighting attempts to break into the webserver.

In general the honeywall will log all packets that pass through it in order to correlate specific streams of packets as an attack. In addition, the

honeywall will often receive data detailing what users do on the local honeypots. Most honeywalls will also have the ability to limit outbound connections from the honeypots in order to prevent a hacker from exploiting a honeypot for denial of service attacks. Combining these features, an administrator is often able to determine the precise method that the hacker employed in order to compromise a honeypot as well as identify the intentions of a hacker based on his or her actions on the honeypot.

Although research honeynets provide an excellent means to track hackers, they suffer from a major drawback: they are essentially passive devices, waiting for hackers to stumble across the honeypots. In a study done by The Honeynet Project [1], 19 unpatched Linux systems had an average life expectancy of 3 months before getting compromised. While this may

be a good phenomena for the average computer user, it becomes difficult to research the latest hacker techniques if attacks occur infrequently. In the same study [1] of Windows honeypots they found that unpatched Windows machines generally have life expectancies measured in hours. However, the short life expectancy of Windows machines was due primarily to worms rather than active hackers, which makes the attacks less valuable as research material.

The results of these previous studies are for research honeypots and it is expected that production honeypots receive significantly more traffic. While we could consider this as one strategy to attract more activity on honeypots, we recognize that usually a company running production honeypots in addition to other production machines will focus more heavily on security than research capabilities. For example, in the event that a company needs to decide whether to sacrifice forensics in order to get a production machine back up and running, we assume that most likely the company will choose to sacrifice the data in order to revive a machine that may be crucial to their business. For this reason we focus mainly on standalone research honeynets and what can be done to make them more attractive.

Overall our goals for research honeynets are

1. Increase number of attacks per unit time
2. Increase overall "quality" of attacks
3. Increase variety of attacks
4. Minimize non-attack background traffic
5. Minimize overall cost of deployment

In the following sections we present strategies that attempt to accomplish these goals.

2.3 Related Research

Lance Spitzner developed the idea of honeytokens [2] that are similar to the concept of honeypots but on a smaller scale. Honeytokens are

files on a system that are not meant to be accessed by anyone, but are developed to stand out if a user is browsing the system looking for interesting files. Accessing a honeytoken triggers an alarm in the system that notifies the supervisor about the illicit behavior, and is meant to be used as a first line of defense against improper insider activity. In this manner honeytokens are meant to be components of an otherwise functional production system, and differ from honeypots that are meant as stand alone systems.

Another similar idea is the Catering Framework [4] designed by Xuxian Jiang and Dongyan Xu that "caters" to the desires of hackers by analyzing network traffic. This framework is designed to dynamically modify honeypots to keep services open that hackers are more likely to use; the Catering Framework makes this distinction by profiling the random network traffic received by outside sources. Any random traffic that is received is assumed to be illicit, and by keeping track of the most prevalent types of network traffic the framework determines what services are best to run on honeypots. While the Catering Framework presents a good strategy for holding onto hackers that find the honeypot, it suffers from the fact that it fails to draw in additional hackers that did not randomly encounter the honeypot in the first place.

Maximillian Dornseif and Sascha May examined models of the cost versus benefit of running a honeynet [3] and found that the cost of running a honeynet can be modeled as $C(t) = S + Mt$ while the utility gained from the honeynet can be expressed as $U(t) = PtM/I$, where S is the initial startup cost, M is the maintenance cost per unit time, P is the amount of utility gained per attack, and I is a factor by which higher investments in the maintenance cost influence the chance of being attacked. Under their model we would like to minimize the overall cost while maximizing the utility gained from the honeynet. We see that in order to do this we would like to minimize S in relation to M . However, their model does not account for methods that arti-

ficially influence the chance of being attacked, which would allow us to increase $U(t)$ while not significantly affecting $C(t)$.

3 Improving Honeynets

3.1 Running a Webserver

The easiest method to increase traffic and the visibility of a machine is to setup a webserver that the outside world can visit. Then by registering a domain name and listing the machine with search engines we can increase the overall traffic on the machine. At first this may result in just an increase in benign traffic, but in the long run it provides hackers with another method through which they can encounter the honeypot.

Here we present a brief description of the steps necessary to setup a more enticing honeypot with a webserver

1. Obtain webserver for operating system of choice

The Apache HTTP Server is one of the most common web servers on the internet due to its powerful feature set, simple installation, and ease to maintain. In addition the Apache HTTP Server is freely available for almost all commonly used operating systems - setup is simple on Windows and most UNIX variants, including Linux, Mac OS and the BSDs. Nearly all package based distributions provide precompiled versions of the HTTP server, but complete sources are available at <http://httpd.apache.org/download.cgi> and can be configured and installed fairly simply on any machines with an ANSI-C compiler.

Several versions of Windows and Mac OS X also come with built in web servers for those adverse to the thought of installing the Apache HTTP Server. Mac OS X's built in web server is apache with a more user friendly interface. Window's built in web server, Internet Information Services (IIS),

is fairly simple to set up and consists of adding virtual directories to the default website through Administrative Tools / Internet Information Services.

2. Building the Website

The next crucial step in attracting hackers is to design a site that has a tendency to draw illicit behavior. While designing an interesting site that brings in a lot of traffic from average internet users is appealing, we do not gain anything from users who visit the site for legitimate purposes. For this reason it is important to design a site that standard internet users have no interest in, but that a hacker would come across when looking for targets.

Money is a typical motivating factor for blackhats, and so we recognize that hackers will be more likely to attempt to break into a honeypot if they believe it will result in monetary gain. An easy method to present this illusion is by designing a site that mimics a financial institution, but with relatively relaxed security measures on the site. If the site is visually well designed and attractive but lacks even basic security measures such as SSL security that many users may not notice (although ideally a blackhat would), it gives the appearance of a relatively inept IT department - an ideal target for a blackhat looking to make money through illicit means.

3. Registering a Domain

Obtaining a domain name is the next step in making a site appear legitimate. There are many different sites on the internet that allow you to register a domain name. After registering the domain name you also need to find a service willing to host your DNS records for you, although, many sites provide a primary DNS server in a package with registering for the domain name. One site that we recommend which provides these services is Yahoo! Domains at

<http://domains.yahoo.com> - they provide a number of tools and DNS servers and only cost 2.99 per year for the domain name.

4. Listing with Search Engines

Unfortunately, many search engines such as google no longer allow you to list your site manually anymore. Instead crawlers automatically prowl the internet for new sites that are linked in from existing ones. Danny Sullivan discusses tips for making websites more visible to search engines in [6]. Some of the biggest tips are to make sure your website is listed in the major website directories and to carefully craft the title / content of the page with regard to certain search terms. The primary directory service that many major search engines use is the open directory project at dmoz.org, and submitting websites is simple using the "suggest URL" feature.

3.2 Hacker chatrooms

Another strategy to increase hacker traffic on honeypots is to go straight to the source - finding the hackers themselves and convincing them to attack your machine. On any decently sized IRC server, for example Undernet, the list of most popular channels includes a number of hacker chatrooms such as #cc-web where the operator advertises rooted machines, credit card numbers, senders, mailers, and hacked ebay accounts. However, as one would not like to reveal to potential hackers the true nature of the honeypots, it is difficult to find ways to coax hackers into attacking your machine without suspicion.

The simplest strategy available is simply to tell the chatrooms that you had a personal machine that you wanted to check the security of - and that you would be glad to have anybody attempt to break into it. Often hackers are motivated by a need for personal acknowledgment, and by presenting a challenge to the hacker you will be acknowledging the hacker's skills if he or she is successful at breaking in to your machine.

Other strategies range from advertising the machine as a valuable box that likely has credit card numbers on it (possibly in conjunction with a webserver set up on the machine) to inciting hackers through insults in order to try and get them to attack the machine in retaliation.

3.3 Obvious Advertising - contests

A third method for developing the traffic on a honeypot is through active advertising such as that done by <http://www.rootthisbox.org/>. Using an ingenious method for attracting traffic, <http://www.rootthisbox.org/> relies on attracting hackers to the site through a challenge - to see who is the best hacker. Machines are submitted to <http://www.rootthisbox.org> and the goal of a number of different teams is to gain root control of as many machines as possible and hold onto that control for as long as they can. Throughout this process, each team is competing against everyone else in what resembles a virtual game. Setting a machine up to act as a honeypot and submitting it to the contest would certainly generate a large amount of research data and benefit the security community greatly. One of the drawbacks of this approach, however, is it relies on the ego and competitive nature of hackers who are trying to show off their skills. I believe that this strategy will attract more "script kiddies" than any other type of hacker because while appealing, blackhats have better things to do with their time than participate in this kind of game. Still, this type of experiment would nonetheless produce interesting and valuable results.

4 Conclusions

Having highlighted a major drawback of honeypots I believe this is an area of important research if we want to fully track the evolving attacks that hackers employ. I have illustrated a number of different methods that one could use to cope with this problem, but there are certainly more methods out there. In future experiments

I would ideally like to test some of these methods, in particular that of setting up an active webserver on a honeypot. Honeynets are still a young technology and as such there are many different experiments that can be done with them. By expanding the rate at which data acquisition is performed on honeynets we essentially expedite all future experiments, which is why I believe this is an important first step in the field of hacker analysis.

References

- [1] The Honeynet Project.
Know Your Enemy - Trend Analysis
<http://project.honeynet.org/papers/trends/life-linux.pdf>.
- [2] Lance Spitner. *Honeypots: Catching the Insider Threat* Annual Computer Science Security Applications Conference, December 2003.
- [3] Maximillian Dornseif, Sascha May. *Modeling the costs and benefits of Honeynets* The Third Annual Workshop on Economics and Information Security, May 2004.
- [4] Xuxian Jiang, Dongyan Xu. *BAIT-TRAP: a Catering Honeypot Framework*
<http://www.cs.purdue.edu/homes/jiangx/collapsar/publications/BaitTrap.pdf>.
- [5] Marc Rogers. *A New Hacker Taxonomy* <http://homes.cerias.purdue.edu/mkr/hacker.doc>, 2000.
- [6] Danny Sullivan. *Search Engine Placement Tips*
<http://searchenginewatch.com/webmasters/article.php/2168021>, 2000.

Author Index

Crosta, Dan, 1

Jones, Heather, 9

Jucovy, Ethan G., 34

Li, Connie, 19

McAvinney, Alan, 27

Parsioan, Taufik, 19

Patton, Kenneth, 45

Prado, Javier, 9

Turner, Ben, 27