Presented at SAFECOMP 2007
Published in Saglietti and Oster
(Eds.): "Computer Safety,
Reliability and Security", LNCS
4680 http://springerlink.com

Survival by Deception

Martin Gilje Jaatun[1], Åsmund Ahlmann Nyre[2], and Jan Tore Sørensen[3]

[1] SINTEF ICT, NO-7465 Trondheim, Norway
Martin.G.Jaatun@sintef.no,
WWW home page: http://www.sintef.com/ict
[2] Q-free ASA
[3] Norwegian Universiy of Science and Technology, NO-7491 Trondheim, Norway

Abstract. A system with a high degree of availability and survivability can be created via service duplication on disparate server platforms, where a compromise via a previously unknown attack is detected by a voting mechanism. However, shutting down the compromised component will inform the attacker that the subversion attempt was unsuccessful, and might lead her to explore other avenues of attack. This paper presents a better solution by transforming the compromised component to a state of honeypot; removing it from duty, while providing the attacker with bogus data. This provides the administrator of the target system with extra time to implement adequate security measures while the attacker is busy "exploiting" the honeypot. As long as the majority of components remain uncompromised, the system continues to deliver service to legitimate users.

1 Introduction

The history of the Internet shows that it is not possible to develop a system that is both *impervious to attack* and *useful* (i.e., provides anything more than rudimentary functionality) – no matter how carefully crafted the armor may be, the vandals[4] always seem to be able to find a chink in it.

Attacks on e-commerce installations and general web sites frequently employ platform-specific exploits based on known vulnerabilities. In later years, the "patch window" has been steadily decreasing, to the point where we now face "zero-day exploits" that are being wielded even before a patch for the specific vulnerability is generally available. Clearly, new mechanisms are required to combat this threat. Our contribution to the cause is a system for *Increasing Survivability by dynamic deployment of Honeypots* (ISH). In the following, we will discuss the theoretical backgound and describe our prototype ISH implementation.

[4] We agree with Marcus J. Ranum [1] that this may be a more descriptive term for what is usually referred to as a "hacker".

2 Background

Protagonists of honeypots have by many been considered the lunatic fringe of the computer security community, but Lance Spitzner [2] and the Honeynet Project [3] have contributed to get more mainstream attention (if not general acceptance) for honeypot ideas. Recent publications such as [4] and [5] at recognized conferences, and others listed on the Honeynet homepage [6] lends academic credibility to the honeypot as an information security resource.

To quote [6], the *raison d'etre* for the honeynet project (and thus, a honeypot) is

> To learn the tools, tactics and motives involved in computer and network attacks, and share the lessons learned.

Indeed, the idea of "peering over the shoulder" of an active hacker has been pursued by many, and classic papers such as [7] and [8] describe how pioneering defenders in an ad-hoc fashion have thrown together what in reality were the first (after-the-fact) honeypot systems[5].

The idea of dynamically transforming a compromised system to a state of honeypot was introduced in [9], but the authors did not describe in detail how this might be accomplished. Disparity and redundancy are classic tenets of dependability [10] and (by extension) survivability. We have employed the definition of survivability given in [11], but with added emphasis on malicious activity rather than accidental incidents.

2.1 Related work

- *SITAR* [12] is an architecture that aims to provide a system invulnerable to attack, using replication, software diversity and a voting mechanism.
- *Bait and Switch Honeypot* [13] is an open source project that is in many ways similar to our own. The main difference is that Bait and Switch uses a firewall proxy to direct malicious traffic to a (permanent) honeypot server, and relies solely on an Intrusion Detection System (Snort [14]) to differentiate legitimate from malicious activity.
- *Shadow Honeypots* [15] also employ a proxy-like mechanism to classify traffic, routing suspicious traffic to a special shadow server that makes a final decision.
- *MPITS* [16] is a relatively simple system that employs disparity and redundancy to offer a basis for intrusion tolerance. We have employed MPITS as an important component in the prototype implementation of ISH; MPITS is described further in section 2.2. Note that ISH depends on service replication on disparate software platforms, but not directly on MPTIS.

2.2 Minimal Proxy for Intrusion Tolerant Systems

MPITS was developed by Broen [16] to provide a less complex basis for intrusion tolerant systems. that additionally would serve as a reference system when comparing existing, more complex systems. Although existing intrusion tolerance systems have a relatively high level of complexity, they are still vulnerable to single point of failure. Acknowledging that the connection point to the external network always will be a single point of failure, MPITS seeks to minimize the likelihood of compromising the unit by limiting the functionality and complexity of the system. The low complexity yields better understanding and enables a more thorough inspection of the source code to eliminate vulnerabilities.

MPITS utilizes replication of services on disparate software platforms to achieve survivability. The system consists of two types of components; a number of application servers and a proxy server (see figure 1). The application servers are the servers containing the actual service the system is providing, while the proxy server works as the connection point to the outside world and manages all inbound and outbound traffic.

The proxy will forward all incoming requests to the application servers, and process the responses. In theory, all well-formed requests should generate the same resonse if the various application servers are functionally equivalent. In practice, there may be minor differences, which is why MPITS groups replies in *equivalence classes*, based on a configurable notion of what is “close enough”. To determine whether two responses belong to the same equivalence class, the responses are compared byte for byte, and all discrepancies counted. If the error ratio is below a configurable threshould, the responses are considered equivalent. Special characters may be weighted to indicate their increased or decreased relative importance, such that numbers may be labeled more crucial than letters in

[5] That these early efforts did not develop further, can partly be ascribed to the fact that this kind of activity quickly proved too time-consuming – a point we will return to later.

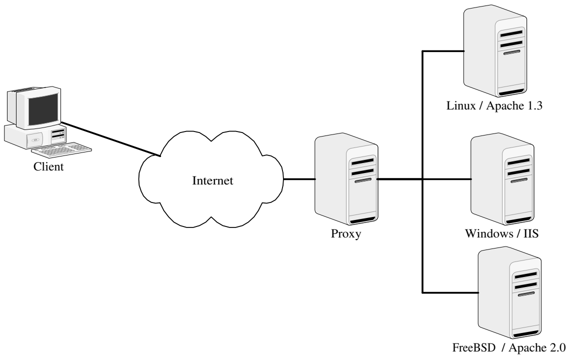

Fig. 1. An overview of the MPITS architecture

a banking transaction. The equivalence algorithm of Broen is rather simplistic, and requires further development. Once the responses in the different equivalence classes have been tallied, MPITS performs a voting process to determine which is the majority response. For the configuration depicted in Fig. 1, the following possibilities exist:

- All three responses are put in the same equivalence class; the request is considered benign, and the response is forwarded to the external client. (Voting: 3-0)
- Two responses are put in one equivalence class, and the last in another; the odd man out is considered compromised, and the response from the majority equivalence class is forwarded to the external client. (Voting: 2-1)
- All three responses are put in different equivalence classes; no determination can be made regarding which response is valid, and the system cannot generate a response. (Voting: 1-1-1; “Hung jury”)

MPITS thus only provides a means for determining that something is amiss, but makes no attempt do do anything about the situation. It is therefore considered a basis or framwork for intrusion tolerance, rather than an intrusion tolerant system.

3 System Idea

The main goals of the ISH system are to deliver critical services to legitimate users even when under attack, detect and detain the attacker without alerting same, and provide bogus data to the attacker. This is illustrated in Fig. 2.

The idea is that if a server unit is exposed to an exploit specific to that particular platform, the response will be different than the one generated by the two other units. All well-formed requests, on the other hand, should result in the same response or output. Once it is determined which unit is the odd man out, this unit can be isolated and removed from further voting.

4 System Overview

A logical description of the ISH system is given in Fig. 3. The fundamental components are as follows:

- Voter
- Router/Switch
- Logging unit
- Proxy
- Server units

In the following, we briefly describe each component.

Voter: The voter component is taken from MPITS, as described earlier. The voter is responsible for detecting attacks based on response discrepancies, and taking appropriate response.

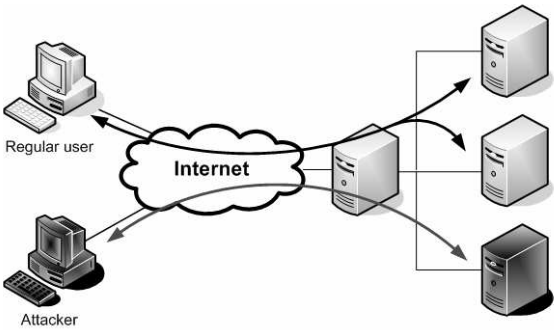

Fig. 2. A compromised component stringing an attacker along

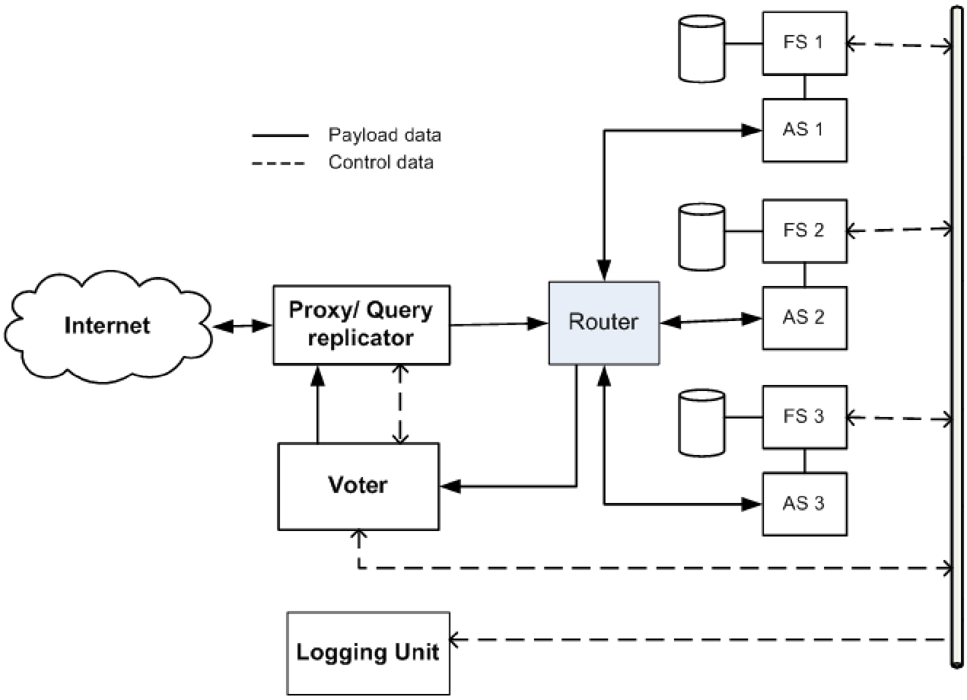

Fig. 3. Conceptual system overview

Router/Switch: For connectivity, and also hiding internal network structure. This is a standard COTS component.

Logging Unit: A separate write-only loghost, for logging attacker activity. The logging is only activated once an attack has been detected.

Proxy: The proxy forwards requests and responses between client and server(s).

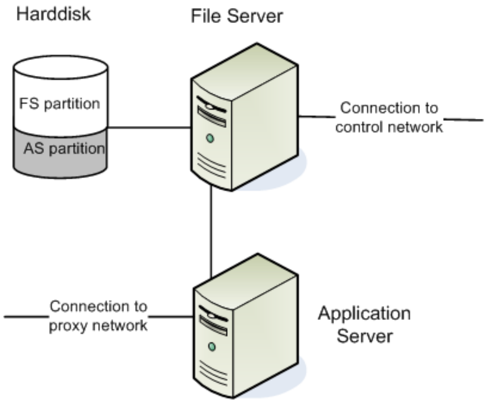

Fig. 4. Server unit

Server Unit: In these days of rootkits [17], it is difficult to trust any system that does not consist of pure hardware. Such systems, however, no longer exist. To up the ante against the attackers, we have designed a prototype server unit that allows us to verify the operating system files of an application server while it is still running.

The server unit is composed of two computers: A file server and the actual application server, as depicted in Fig. 4. The latter has no writeable file system of its own, but mounts an exported file system from the former. In addition to providing the application server with a file system, the file server also performs integrity checking of important files [18] to detect a compromise, and will report any anomalies to the voter.

Once a compromise is detected (either via the voter, or via the file integrity check), the file server will replace all sensitve data on the unit with obfuscated data prepared in advance. This is accomplished by maintaining a shadow volume that is periodically updated to keep it roughly equivalent to the real partition; the file server can then simply switch partitions – in all but the most unfortunate circumstances[6] this would be possible to achieve without the attacker noticing.

The dedicated file server could in theory be replaced by a Storage Area Network solution where an independent mechanism could verify the integrity of the files.

5 Implementation

For practical reasons we had to scale down our ambitions for the initial protoype implementation; while the original plan had called for three separate hardware/software platforms, e.g. Solaris on Sun, Windows on Intel and Linux on PowerPC, we had to settle for identical Intel PCs running FreeBSD and Linux, and two different versions of the Apache web server. The prototype ISH implementation is illustrated in Fig. 5.

The proxy unit is based on MPITS, extended with an attacker handling feature. Most importantly, when the voter detects a 2-1 voting anomaly, we no longer forward the majority response to the client; in this situation the client is assumed to ba an attacker, and thus the minority response (from the compromised unit) is returned.

5.1 Identifying and Detaining the Attacker

Based on our description above, detecting that an attack has taken place based either on a voting discrepancy or on file integrity violation detection is fairly manageable (but see also section 5.2). However, new challenges arise when faced with the problem of identifying the attacker.

If we operated a single-user system, it would have been trivial: The current user would have been the culprit. With several concurrent users, it is more difficult – we have settled for labeling the first request causing a discrepancy in the voter as an attack, and using the originating IP address of this request as the address of the attacker. We acknowledge that this approach has its obvious downsides due to the common usage of VPNs, NATs and the fact that attackers may control several IP-addresses through bot-nets or the like. If the server utilizes a login feature, a user profile may be utilized, but for servers with open access such as web servers, this approach is not feasible.

Once the attacker has been identified, traffic originating from this source is no longer distributed to all the units; only to the compromised unit (now acting as honeypot). Responses from the honeypot is naturally not voted on, but passed on directly. In this sense, the user (i.e. attacker) interacting with the honeypot

[6] In the case of a web server with dynamic content, such an unfortunate circumstance could be that some information has significantly changed or been added just prior to the attack; the attacker might then notice that the information is different or missing on the "compromised" unit. This only applies to information that would be available to all clients, and not to information that first requires a breach of the access control mechanisms of the service.

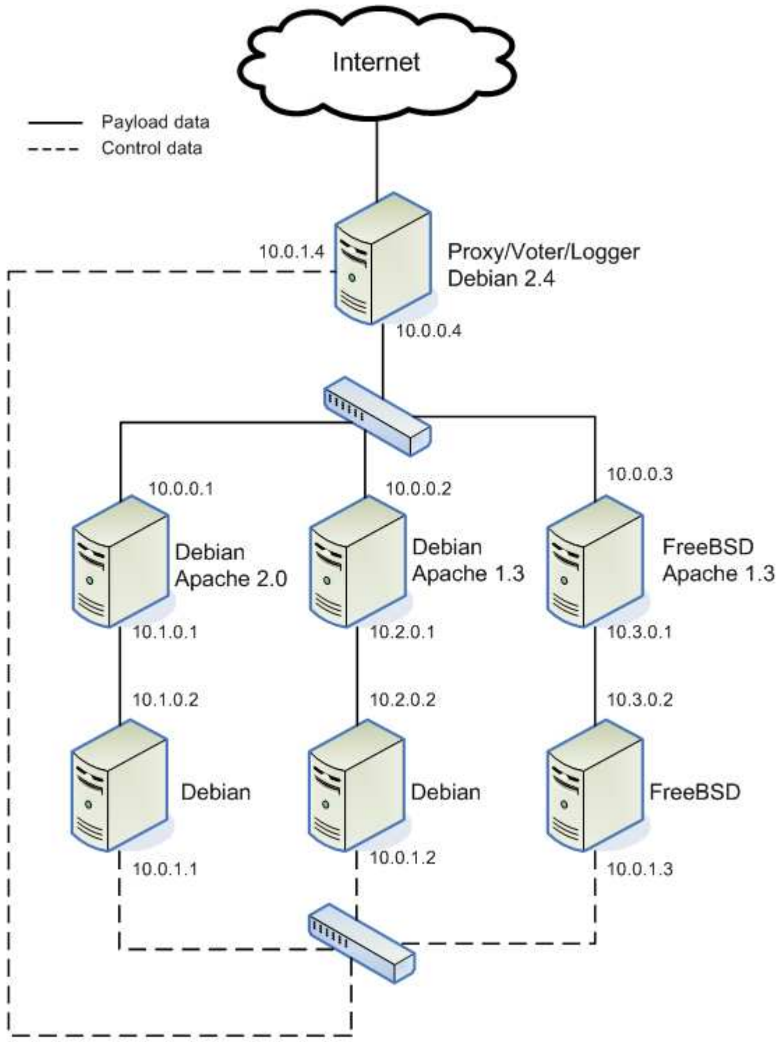

Fig. 5. Prototype System

actually experiences improved efficiency with respect to an uncompromised system; this “spare capacity” may be used e.g. for extended logging of attacker activity.

5.2 False Positives and Negatives

The success of the voter relies on the assumption that well-formed input will result in the same output, regardless of which platform the service is implemented on. Unfortunately, practical tests have shown that this is not necessarily true, in rare cases causing the voting unit to detect a discrepancy based on legitimate input. This implies that careful configuration is required on each platform, and special modifications (e.g. suppresion of platform-specific staus messages) may be necessary to prevent spurious discrepancies. This of course also has implications for the specification of equivalence classes, as mentioned earlier.

Furthermore, it is possible that a given exploit would generate innocuous output, but have side-effects that causes a compromise further down the line. Since the voter is based on the generated output, such an attack would not be detected.

6 Discussion

Strictly speaking, dynamically transforming a system into a honeypot only makes sense as long as you are dealing with a human attacker; i.e. if you are concerned with detecting “dumb”, indiscriminate network worms, a few ordinary honeypots sprinkeled around your domain would probably have served nicely. However, even in such cases the service duplication of our system will ensure that the adverse effects of the worm will be mitigated.

Another objection to ISH might be that it represents “security through obscurity” – one could argue that once it is known that a given installation is implemented by ISH, its value is nil. However, if ISH were to be deployed in the spirit of “Defense in depth” we claim that this does represent at least two additional lines of defense; no single-platform exploits would be applicable to our system, and any attack that does succeed, but changes one of the files in our “integrity check set” will still be flagged and cause the unit to be removed from service.

To focus on our idea of dynamic heneypot deployment, we have in our presentation intentionally omitted discussing other network intrusion detection mechanisms (e.g. cite [19], [20]), but we acknowledge that to the extent that ISH is an intrusion detection tool, it may be augmented by employing additional (traditional) IDS mechanisms, either before the traffic reaches ISH, or as part of ISH.

6.1 Determining the optimal number of units

The use of three server units in our prototype means that once a single unit is transformed into a honeypot, a subsequent compromise of one of the two other units can be detected but not localised. This could be remedied by increasing the number of units[7] , but there will none the less be a point at which additional intrusions will mean that the entire system must be taken down. However, as long as systems are diligently updated with the latest patches (and otherwise protected against known attacks), such an occurence will be rare – zero-day exploits aren't *that* prolific. Also note that as long as the server units are on disparate platforms, repeat infections from automatic "bots" will be avoided, as only the unit that already is compromised will be vulnerable to a given exploit[8].

Also note that in contrast to a traditional honeypot, the goals of ISH do *not* include being penetrated by known exploits – we assume that in a production system, other measures will be in place that will be able to block known attacks. Thus, only new, otherwise undetected, and *successful*[9] attacks will cause ISH to transform a unit to honeypot state.

6.2 Application Areas

ISH would be applicable to business-critical services with high availability and security requirements. However, we realize that network administrators already have their hands full with managing the current crop of firewalls and intrusion detection systems. Furthermore, the ISH system also represents added cost for hardware (in the worst case multiplying the initial procurement costs by seven), software development (at least three-fold) and maintenance.

Thus, we presume that ISH would be of greatest interest to *managed security providers* for customers who offer such business-critical services to *their* customers. The managed security provider could deploy ISH to provide the service in question, but would additionally use it as a complement to their existing efforts in detecting new attacks/exploits. This would be of benefit not only to the current customer, but also to other customers of the managed service provider and to the community in general.

For the managed security provider, ISH would represent an improvement over a conventional honeypot or honeynet, in that the deployed ISH system would be a "real" system until it is successfully attacked.

7 Further work

Our prototype ISH implementation is a very simple web server; a logical extension would be to implement a system for a generic service. There are also ample opportunities for less trivial extensions.

[7] or by using virutal machines

[8] To be fair, this would ultimately require all the software to be devoloped according to "true" n-version programming [21].

[9] Attacks that are unsuccessful, either due to blocking by other mechanisms, or because the underlying system is not vulnerable to the particular attack, will not trigger honeypot deployment.

7.1 Code Integrity

Even though we practice defense in depth to detect intruders who do not trigger a voting anomaly, there still remains the challenge of detecting intrusions that do not alter the file system of the affected unit.

In a very interesting approach presented by Wang and Dasgupta [22], it is possible to verify the correctness of all static parts of a Linux kernel by employing a special autonomus "co-computer" (a single-board computer with access to the system bus). We believe this solution could be adapted to ISH to increase protection against arracks that leave the file system intact.

7.2 Attacker Separation

It would have been preferable to identify the attacker based solely on the session generating the exploit traffic, and used dynamically configured switches to route traffic from/to the attacker along a separate path. This would also allow us to separate an attacker from legitimate traffic originating from the attacker's IP address.

7.3 Darkhost Voter

Although we have strived to keep the proxy/voting unit simple, it still has an uncomfortable level of complexity when considering that it represents a single point of failure with respect to security.

If we can put the voting mechanism on an "invisible" (i.e. dark) host without an IP address, we leave only a very simple proxy and query replicator on the publicly available host, while at the same time ensuring that the vital (and much more complex) voting unit cannot be adressed directly from the internet. The darkhost would have to craft packets with the proxy unit as originating address; this would require some fancy footwork with respect to ensuring that TCP sequence numbers etc. are properly maintained.

8 Conclusion

We have presented ISH, a prototype system that through duplication of server units detects new platform-specific attacks, and enhances the survivability of the system as a whole by transforming the compromised unit to a state of honeypot, while the uncompromised units continue to deliver service to legitimate users.

Acknowledgements

We are very grateful to Torgeir Broen for being allowed to use the MPITS code base as a starting point for our own efforts.

References

1. M. J. Ranum, “Thinking about firewalls,” in *Proceedings of Second International Conference on Systems andNetwork Security andManagement (SANS-II)*, April 1994.
2. L. Spitzner, *Honeypots – Tracking Hackers.* Addison-Wesley, 2003, ISBN 0-321-10895-7.
3. The Honeynet Project, *Know Your Enemy – Revealing the Security Tools, Tactics and Motives of the Blachat Community.* Addison-Wesley, 2002, ISBN 0-201-74613-1.
4. P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, “The nepenthes platform: An efficient approach to collect malware,” in *Recent Advances in Intrusion Detection, Proceedings*, ser. Lecture Notes in Computer Science, 2006, vol. 4219, pp. 165–184.
5. F. Pouget and T. Holz, “A pointillist approach for comparing honeypots,” in *Detection of Intrusions and Malware, and Vulnerability Assessment, Proceedings*, ser. Lecture Notes in Computer Science, 2005, vol. 3548, pp. 51–68.
6. “The HoneyNet Project.” [Online]. Available: http://www.honeynet.org
7. B. Cheswick, “An evening with Berferd in which a cracker is lured, endured, and studied,” in *USENIX Conference Proceedings.* USENIX, 1992, pp. 163 – 174.
8. C. Stoll, “Stalking the wily hacker,” *Communications of the ACM*, vol. 31, no. 5, pp. 484–497, May 1988.
9. M. G. Jaatun and G. Hallingstad, “Techniques for increasing survivability in NATO CIS,” in *proceedings of the 1st European Survivability Workshop*, Köln-Wahn, Germany, Feb. 2002.
10. J.-C. Laprie, “Dependable computing and fault-tolerance : Concepts and terminology,” in *proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, 1985, pp. 2–11.
11. R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, and N. R. Mead, “Survivable network systems: An emerging discipline,” Sofware Engineering Institute (SEI), Carnegie Mellon University, Tech. Rep. CMU/SEI-97-TR-013, 1997-1999.
12. F. Wang, F. Gong, C. Sargor, Goseva-Popstojana, K. Trivedi, and F. Jou, “SITAR - a Scalable Intrusion-Tolerant Architecture for Distributed Services,” in *proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, United States Military Academy, West Point, NY, June 2001, pp. 38–45.
13. “Bait and Switch Honeypot.” [Online]. Available: http://baitnswitch.sourceforge.net/
14. “Snort - a network intrusion detection system.” [Online]. Available: http://www.snort.org
15. K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis, “Detecting targeted attacks using shadow honeypots,” ICS-FORTH, Crete, Greece, Tech. Rep. TR-348, Jan. 2005.
16. T. Broen, “Innbruddstolerante systemer: En eksperimentell utprøving og vurdering,” Master’s thesis, University of Oslo, Norway, May 2005.
17. G. Hoglund and J. Butler, *Rootkits*, 1st ed. Addison-Wesley, 2006.
18. Samhain Labs, “The Samhain file integrity system user manual,” available from http://la-samhna.de/samhain/manual.
19. R. G. Bace, *Intrusion Detection.* Macmillian Technical Publishing, 2000.

20. S. Northcutt and J. Novak, *Network Intrusion Detection*, 3rd ed. New Riders Publishing, 2002.
21. N. Ashrafi, O. Berman, and M. Cutler, “Optimal-design of large software-systems using n-version programming,” *Ieee Transactions on Reliability*, vol. 43, no. 2, pp. 344–350, 1994.
22. L. Wang and P. Dasgupta, “Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System,” in *Proceedings of the Third IEEE International Symposium on Security in Networks and Distributed Systems (to appear)*, 2007.