2012

Securely Handling Inter-Application Connection Credentials

Gary Lieberman
Nova Southeastern University, gary@lieberman.us

Securely Handling Inter-Application Connection Credentials

By
Gary Lieberman

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Information Systems

Graduate School of Computer and Information Sciences
Nova Southeastern University

2012

We hereby certify that this dissertation, submitted by Gary Lieberman, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

[illegible] Sept 12, 2012

Francisco J. Mitropoulos, Ph.D. Date
Chairperson of Dissertation Committee

[illegible] Sept 12, 2012

Eric S. Ackerman, Ph.D. Date
Dissertation Committee Member

[illegible] Sept 12, 2012

Gregory E. Simco, Ph.D. Date
Dissertation Committee Member

Approved:

[illegible] Sept 12, 2012

Eric S. Ackerman, Ph.D. Date
Interim Dean, Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences
Nova Southeastern University

2012

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Securely Handling Inter-Application Connection Credentials

By
Gary Lieberman
September 2012

The utilization of application-to-application (A2A) credentials within interpretive language scripts and application code has long been a security risk. The quandaries being how to protect and secure the credentials handled in the main body of code and avoid exploitation from rogue programmers, system administrators and other users with authorized high levels of privilege.

Researchers report that A2A credentials cannot be protected and that there is no way to reduce the risk of the inevitable successful attack and subsequent exploit. Therefore, research efforts to date have primarily been focused on mitigating the impact of the attack rather than finding ways to reduce the attack surface.

The work contained herein successfully addresses this serious cross-cutting concern and proves that it is in fact possible to significantly reduce the risk of attack. This reduction of risk was accomplished through implementing a method of credential obfuscation which applied advice with concerns utilizing a composition filter. The filter modified messages containing the credentials as they were sent from the interpretive language script to the remote data store.

The modification extracted credentials from a secure password vault and inserted them into the message being sent to the remote data store. This modification moved the handling of the credentials from the main body of code to a secure library and out of the reach of attackers with authorized high levels of privilege. The relocation of the credential handling code lines significantly reduced the attack surface and the overall risk of attack.

Acknowledgments

First and foremost I would like to thank my wife Carol for her encouragement and undying support. Without which I would never have had the courage and fortitude to persevere in graduate and post-graduate school. A deep thank you also goes out to my brother-in-law Phillip Stein, PhD whose accomplishments I have always admired. Your many encouraging talks have made such a difference.

My appreciation and love goes out to my children, Mark and Sarah who graciously accepted the many hours dad was doing school work and not participating in family things. Even though they are adults with successful careers, family is still most important to them and their sacrifice is very much appreciated. Sarah, thank you for the endless hours of boring proofreading my graduate and post-graduate papers. I have no idea what my papers would have read like without your help.

Dr. James Cannady, thank you for the support and encouragement when you told me I had a good idea in my first semester. Your introduction to Dr. Mitropoulos was key. Dr. Eric S. Ackerman, I really appreciated our chats in your office. It was really nice to have that open access to the dean. I was truly honored when you agreed to serve on my committee. Dr. Gregory Simco, arguably the toughest grader I've ever encountered, everything you said was dead-on correct. You made me a better writer and an even better professor. Lastly, a very special thank you goes to Dr. Frank Mitropoulos for your guidance and support. The first time I met you I said there's my committee chair. Your guidance has been invaluable.

To everyone who supported my effort, both mentioned and not mentioned; I couldn't have done without you!

This work is dedicated to my father, Alfred Louis Lieberman, an amazing role model.

Table of Contents

List of Tables

List of Figures

Chapter 1

What can we take on trust in this uncertain life? Happiness, greatness, pride – nothing is secure, nothing keeps.
– Euripides, *Hecuba* (c. 425 B.C.)

Introduction

Background

The utilization of application-to-application credentials within interpretive language scripts and application code has long been a security trade-off. The quandary being should the credentials be embedded in the code and risk exploitation from rogue programmers, system administrators and other users with authorized high levels of privilege or should the application not require password authentication at all? The latter can be accomplished, somewhat securely, by running the entire process on a single highly audited machine, disconnecting it from the network and isolating the machine from all but a few trusted individuals. Although this approach is commonplace in top secret government processing it is highly impractical in a commercial data processing environment. Therefore, IT security experts, standards bodies and auditors alike have concentrated their efforts on figuring out new and ingenious ways to mitigate the impact from data loss rather than researching methods for securing the credentials. The research presented in this paper focuses on the development of a method to address this gap and significantly reduce the risk of credential exploitation. It would be naive to think that the risk can be eliminated entirely. No credential handling method, process or system can claim to be totally secure, but risk can be significantly reduced to the point where batch processes utilizing interpretive language scripts can be considered secure in a relative sense.

This document is organized as follows; the Problem Statement section defines the problem being addressed and qualifies it as being research worthy. It is followed by a description of the dissertation goal. Following this section is the Relevance and Significance section which provides support for the problem statement and dissertation goal sections and addresses the questions: why there is a problem and who is impacted? The Literature Review section builds a foundation supporting the validity of this research. Supporting evidence in this research area is gathered from the existing body of knowledge and is critically reviewed and analyzed. Additional supporting evidence is gathered from federal laws and regulatory agencies, established auditing standards and generally-accepted professional guidelines and practices. Concluding the Literature Review is an analysis of patent applications and third party products that attempt to address this problem. The final sections present the research, analysis of findings and conclusions.

Problem Statement

The Association of Certified Fraud Examiners reported in their 2010 Report to the Nations on Occupational Fraud and Abuse (as cited in Nilsen, 2010) that on average companies lose 5% of their revenue to fraud and abuse. Furthermore, researchers have found that 80% of the fraud and malicious activities within IT operations are related to the misuse of passwords (Singleton, 2002). According to a recent Gartner report, close to

90% of all software attacks are aimed at the application layer (as cited in Brandel, 2009), yet the 2010 Global Security Survey of Financial Institutions by Deloitte and Touche ("2010 Financial Services Global Security Study," 2010) reported that only 46% of all survey respondents include application security in their software development life cycle and that 23% of all respondents lack well-defined security policies in their development standards. It was reported by the respondents that 10% of all data loss was from malicious software installed by employees with proper access and authentication. Over 70% of those surveyed have a medium to high expectation of attacks on program source code. Lastly, only a disturbing 39% of the respondents reported having a high level of confidence in their ability to defend against insider cyber-attacks.

To better understand the relevance of these survey statistics one must consider that 70% of all insider attacks stem from software exploits ("Insider threat study: illicit cyber activity in the banking and finance sector," 2004) and that those vulnerabilities primarily lie within the context of software development and deployment. A primary area of concern is the manner in which the connection credentials that are used to establish a communication channel between two applications such as a client and a database server are handled.

Randazzo, Keeney, Kowalski, Cappelli and Moore (2005) found that 17% of all insider attacks came from administrators with legitimate privileged access rights. Baring in mind that administrators may not always be trustworthy (Jaeger & Tidswell, 2001) it is a fairly simple task for a system administrator of questionable repute to harvest hard-coded passwords from batch processing scripts and/or application programs stored on the file systems that they support. It is an equally simple task for a rogue administrator or software developer to modify a script and cause the password to be exported as it is received from an associated password vault.

Chinchani, Iyer, Ngo and Upadhyaya (2005) found that a lack of a practical methodology has security experts believing that insider attacks are unpreventable. They state that the only practical solution is to increase system monitoring, application logging and overall security countermeasures. They further admit that these are all highly inconvenient measures that foster an atmosphere of distrust within the organization's IT staff. After conceding that little can be done to prevent such attacks they concentrate on presenting an enhanced threat modeling framework to help identify those applications which are most vulnerable and good candidates for increased vigilance. This is a reactive approach that highly depends on both successful monitoring and detection of the attack, and more importantly, successful interception of the data before it leaves the premises. A better approach would be to work towards preventing the attack from happening altogether. Unfortunately, few research projects in this area recommend approaches aimed at addressing insider attacks proactively (Cappelli, Moore, Shimeall, & Trzeciak, 2009).

Shiflett (2004) emphatically warns that there is no way to hide or keep connection credentials safe from rogue privileged users. He proposes a simple solution in which connection credentials are stored in environment variables and accessed by the script at the time of execution. This method seems rather naive as environment variables must be stored as cleartext in files on the same system as the scripts themselves. They are just as vulnerable to the same privileged user attacks as the scripts themselves. In addition, all memory-stored environment variables are accessible by the program code and therefore vulnerable to rogue programmer attacks. Lastly, operating systems such as certain UNIX variants HP/UX, Solaris, LINUX, MAC OS X and others allow for the reporting of an executing process' operating environment. Thus, memory-stored environment variables are easily viewed by all users of privilege on those systems.

Chumash and Yao (2009) also believe that due to the interpretive nature of scripts, it is impossible to obscure, mask or hide connection credentials from users with administrative access. They identify the most common solution in current use which is to store the connection credentials as cleartext in files external to the script itself in a location outside of the script's root file system and to include or source them in during the script's execution. Their thinking is that the credential's location could be better monitored and protected than that of the script itself. This methodology is short-sighted and falls prey to the same privileged user attack as if the credentials were stored in the script itself. To address this concern they propose what is essentially a password vault which encrypts and stores the credentials in a secure environment and returns them to the executing script once it has been properly authenticated. Although this method removes the credentials from viewing by people with read access to the script it does not prevent people with write access, such as administrators, from modifying the script and capturing the returned credentials. There is also exposure to rogue programmers who may not have access to the production system that executes the script but who do have access to the source code and can place a logic bomb in the code that will capture the passwords and forward the credentials to awaiting accomplices when the script executes. This negates almost all security measures on the executing system, as the attack is setup and put in place prior to the software rollout.

Further exacerbating matters, Said, Guimaraes, Maamar and Jololian (2009) found that there is a total lack of literature dealing with application security specifically in areas concerning application connection credentials. Because of this they propose to indoctrinate programmers and software developers with security courses in graduate school curriculums. Of the 83 institutions they surveyed only 10 had such security courses. The fact that research institutions and academia in general are not recognizing the cleartext storage of connection credentials as a legitimate risk and wide-spread problem further supports the need for more research and literature on this subject. Educating the development staff is a solid step in the right direction, but it does not address the pressure to shortcut good development standards and circumvent security reviews due to understaffing and development cost considerations. Further, it does not address in any manner, shape or form the corruptibility of administrators and software developers, especially in times of an economic downturn.

Shmueli, Vaisenberg, Elovici and Glezer (2010) tell us that access control is useless if the attacker is a system administrator or a database administrator. They propose a database encryption scheme to make the data stored in the database unreadable when the inevitably successful insider attack happens. Besides being quite pessimistic in nature, their solution assumes the attacker will access the database with the credentials of the database administrator to look at the stored data. They do not address the possibility of the rogue administrator capturing the connection credentials from a legitimate batch processing script and accessing the decrypted data with the appropriate level of access authentication afforded the script.

It is typical for many researchers to concentrate on securing the credential repository or fortifying the trust negotiation process and ignore the security of the software that negotiates the trust itself. An example of this can be found in the Yu, Winslett and Seamons paper on automated trust negotiation strategies which clearly and emphatically tells us that they assumed that all trust negotiation software can be trusted (Yu, Winslett, & Seamons, 2003). Six years later Yu, Sivasubramanian and Xie point out that traditional access control is embedded in application code and therefore it is impossible to test its effectiveness (Yu, Sivasubramanian, & Xie, 2009). They also state that it is impossible to protect that application's code from an insider attack.

Franqueira, Cleeff, Eck and Wieringa (2010) find that few approaches consider the proactive prevention of insider attacks and adopt a forensic approach analyzing data gathered from security monitoring tool logs. This seems to be the most common approach proposed by researchers. Franqueira, Cleeff, Eck and Wieringa concentrate their study on what they call "external insiders" and propose that external service providers and their contracting customers bolster contractual agreements to require a high level of log data sharing so that the log analysis is not concentrated in one company or within a single office or location. As with other similar forensic approaches this is closing the barn door after the animals have escaped. The better approach is a prophylactic one in which access to the credentials is restricted to the interpretive script and not the administrator.

The presence of application-to-application connection credentials within interpretive language scripts presents a significant risk, leaving those credentials open to being attacked and exploited. To date no methodology or framework exists that effectively secures those credentials and reduces that risk.

Dissertation Goal

Analysis of research efforts, auditing and regulatory agency standards and third party products tells us that most researchers have conceded that the problem as stated in the above Problem Statement section is unsolvable. They have directed their research and recommendations to be more in line with an approach that attempts to detect the attack after the fact and mitigate the impact of the data loss rather reduce the risk of attack altogether (Salem, Hershkop, & Stolfo, 2008). Albeit naive to state that all risk can be eliminated, this research shows that the risk of insider attack and exploitation of application-to-application credentials can be significantly reduced as measured by the delta between a pre-and post-risk analysis matrix.

The method that was used for comparative analysis to measure the delta between the pre-and post-analysis of risk is a quantitative Decision Analysis Methodology. Kepner and Tregoe (1981) defined the three primary elements of analyzing alternatives and making good decisions as the quality of the problem definition, the quality of the evaluation of alternatives and understanding the impact of the alternatives. They suggest that any decision process can be reduced to a mathematical formula resulting in a weighted quantitative index of alternatives against which an informed analysis can be made. To facilitate this Kepner and Tregoe introduced the concept of a Weighted Decision Analysis Matrix.

For the purposes of quantifying pre-and post-research password exploitation risk, the concept put forth in Kepner and Tregoe's Weighted Decision Analysis Matrix theory was used. The required matrix consists of an array presenting pre-and post-research tests on the vertical axis. The horizontal axis represents a list of attacks that were performed against the test scripts attempting to mine the password used in the scripts. Additional detail on attack vector quantification and measurement ideology can be found in the "Quantifying Testing Results" section in Chapter 3.

This research established a methodology and framework by which an interpretive language script, such as one written in Perl, can securely call a subroutine or function establishing a connection to a data store or secondary application. The request can then be authenticated and have a connection handle returned to the calling function from the called subroutine or function. The connection process was performed in such a manner as to significantly limit the ability of rogue privileged users or rogue developers from trapping the connection credentials for exploitation. In addition to avoiding privileged user exploit threats, this framework also avoided granting the development and application support staff knowledge of the connection credentials which presents an equivalent risk of exploitation.

Relevance and Significance

The cleartext hard-coding of application-to-application connection credentials presents an industry-wide security software weakness that originates in the development process. The weakness then travels with the software into production, leaving the production processing environment vulnerable to attack. Logic Bomb attacks and Trojan Horse attacks can quite easily exploit credentials stored within interpretive scripts (Yang, 2009). The risk is further exacerbated when one considers that the developers of the software often are aware of the credentials when they embed them in the code. These are the same developers that standards bodies and audit reviews recommend having their access to production systems denied for security purposes.

Further support for this as an industry-wide software development issue is bolstered by the listing of hard-coded credentials as a dangerous vulnerability in most security and auditing standards. For example, the Open Web Application Security Project (OWASP), arguably one of the major players in the field of secure software development (Futcher & Solms, 2008), lists hard-coded application credentials eighth on its list of top 10 application vulnerabilities (OWASP, 2007). The SANS Institute, a prominent for-profit research and education organization, lists hard-coded credentials 21st on its list of the top 25 Most Dangerous Programming Errors (B. Martin, Brown, & Paller, 2009). Unfortunately, the most common method to address these issues is a forensic one rather than a prophylactic approach (Salem et al., 2008).

The primary reason so many researchers adopt a forensic approach to their research is the pervasive belief and subsequent acceptance that insider attacks on software and application-to-application credentials are unavoidable and unstoppable. Researchers such as Blackwell (2009); Chumash and Yao (2009); Franqueira, Cleeff, Eck and Wieringa (2010); Shiflett (2004) and Shmueli, Vaisenberg, Elovici and Glezer (2010) have all based their papers on the acceptance of the inevitable security breach initiated by the privileged user. The research presented in this paper was aimed at significantly reducing the risk of insider attack on application-to-application credentials and subsequently changing this perception.

Barriers and Issues

The goal of this research was to develop a methodology by which applications that utilize interpretive scripting languages can be secured against the introduction of erroneous or intentional code logic that could open backdoors and establishes covert access points to production systems. The hard-coding of production application credentials in cleartext into source code exposes those credentials to all development and administrative personnel with access to the development environment, thereby granting defacto access by development and administrative personnel to production processing systems. In addition, this research has developed a methodology for securely handling application connection credentials returned to software applications from a password vault. Specifically, instructions can be embedded into the logic of a program to exploit the returned credentials by forwarding them to a person or persons who would otherwise not be granted access to the targeted production system.

To date adequate and functional solutions to these issues have not been available to the software development community (Boström, 2004; Edge & Mitropoulos, 2009). The proliferation of password vault technology is a step in the right direction, but does not offer the complete solution. Therefore, regulatory agencies, standards bodies and auditing practices have graciously accepted that there is no complete solution and adopted a segregation of duties approach which reduces risk. However, this does not mitigate the risk altogether and is not a complete solution. We can only surmise that because of this wide acceptance of risk by the professional user community funding is lacking for academic research. Furthermore, a lack of funding could be a contributing factor in the lack of peer-reviewed papers proposing solutions. With the exception of the

2004 De Win, Joosen and Piessens case study on FTP Server access controls, very few studies have presented practical approaches to securing application-to-application connection credentials (Boström, 2004; Edge & Mitropoulos, 2009). However, the existence of numerous commercial password vault offerings and the ever-growing number of patent applications attempting to secure the returned credentials seems to support that research in this area is viable and is taking place.

Summary

The position held by researchers that nothing can be done to reduce the risk to application-to-application credentials used in interpretive language scripts is unfortunate. Even more unfortunate is that research is now focused on mitigating the impact of attack exploits that are considered inevitable. This phenomenon is exacerbated by the acceptance by standards bodies and auditing societies who recommend segregation of duties as the primary defense for handling attacks coupled with enhanced monitoring tools that reduce the time between attack and discovery. Better monitoring leads to faster discovery, which lends itself to reduced exploit impact. The hypothesis that it is impossible to reduce the risk of exploit to application-to-application credential in interpretive language scripts is wrong. The research presented in this work proves that the risk can be significantly reduced.

Chapter 2

Review of the Literature

In 1990 the Expect language was introduced by Don Libes (1990). Expect enables two programs that would normally interact with humans to interact with each other. In his paper he clearly states that "Using Expect, it is possible to create a script that solves the passwd problem." He is referring to using Expect scripts to hold passwords and submit them to other applications that would normally require an interactive password submission. Arguably, Libes' work is the foundation of modern day batch processing. He fails to mention, or even consider, the risk of embedding credentials in a script. Later on in 1993 he published his second paper where he introduces Kibitz, which is an Expect add-on that is platform independent and seems to correct the password-embedding issue by allowing the script to ask the executing human for the password (Libes, 1993). Libes, however, states that this password-asking ability is for portability and never considers it for security. He reasons that the script may need to connect to multiple targets using different passwords. Therefore, storing a password in the script becomes inconvenient.

In 1994 Libes published a third paper entitled "Handling Passwords with Security and Reliability in Background Processes" in which he presents five techniques for using the Expect language (Libes, 1994a). The last three techniques are centered on hard-coding the password in the Expect script. He addressed the security concerns by proposing the scripts be permissioned so that only root level privileges can access them. No mind is paid to consider the risk of exploit from those possessing root level privileges on the system. Later that year Libes published another paper moving Expect into the X-

Windows environment. Again, he extolls the virtues of using Expect to handle embedded credentials and never considers the risks (Libes, 1994b).

Libes' one mention of security in the Kibitz add-on, which asks the user for the password, works well in theory. However it breaks down quickly when one considers the large number of batch processing scripts that run each night processing data from the day's business. Such is the case in brokerage houses, banks and other financial institutions. By its nature a batch processing script is designed to run automatically and unattended. Requiring manual intervention in the running of hundreds or thousands of batch processing scripts is nearly impossible and presents a security breach unto itself.

Two years after Libes published his last work on the subject, Mavrikidis (1996) warned against hard-coding passwords in MVS and UNIX scripts and proposed using a password repository to store the passwords so they cannot be harvested from the source code. The author also stressed the need for intrusion detection systems and log monitoring as a final step in protecting the processing environment. Unfortunately, no attention was paid to the rogue system administrator and/or rogue software developer threat.

In their 2009 paper, Englert and Shah (2009) identify the need to protect application credentials against attacks. The authors realize that most access to computers and computer programs is via an ID and password. The goal of their research was to develop an online safe or vault that allows users to securely store and retrieve their passwords for use when interacting with Internet-facing applications. Although their solution does address numerous weaknesses found in online password vaults, it is designed primarily to be used interactively with a human being. Their solution addresses the security of the storage of the password and its eventual transmission to the requesting end user. However, in their conclusion they propose that the same methodology could be used to develop an application-to-application API. If they were to move forward with this expansion of their work they would have to consider the security of the returned credentials once extracted from the vault and safely returned to the requesting application. However, since no credence was paid to the handling of the credentials once returned to the human requestor in their current research, it is unlikely it will be considered in their future research.

Boyen attempts to address the problem of securing credential storage and retrieval in his paper entitled “Hidden Credential Retrieval from a Reusable Password” (2009). He proposes a credential retrieval protocol that includes handshake, secure transfer and decryption in a single protocol operation that returns no indication of success or failure. Successful authentication generates the return of a plaintext password; failure returns a string that could be a password. Thus an attacker using a brute force repetitive attack would gain no insight as to the success or failure of each attempt. Passwords are stored via a process that includes the generation of a random private signing key, the hashing of the password and a signature. All are then transmitted to the server over a secure transmission channel. The passwords are not stored in plaintext on the server as is the case with some password vaults. By storing an encrypted version of the password the author addresses the threat of insider attacks on the storage server itself. Little attention is paid to the threat of insider attacks at the receiving end of the password retrieval request.

Boyen’s work is pointed at the forgetful human being who has a tendency to write passwords down. However, because his proposed password vault is accessible via a network connection the interaction with the vault’s interface could be adapted for use with a batch process using an interpretive language script such as Perl. Boyen did not consider this possibility when analyzing the success of his work or in any proposed future work.

Zhu, Feng and Chen (2009) advise that advancements in computer security such as facial and fingerprint recognition help protect human-to-application credentials, but does nothing to protect application-to-application credentials from exploit by people with administrative level authorization. To address this they suggest that access control policies based on behavior patterns be implemented. Studying these patterns would allow for the identification of computer system users through the similarity of the user's behavioral patterns. Constructing access control policies based on usage patterns has quite a bit of merit and allows for a more granular and finely-tuned set of policies. Although the authors claim this to be an effective means of combating insider attacks, they neglect to address insider attacks perpetrated by users with justifiable privileged access.

Kostiainen, Ekberg, Asokan, and Rantala (2009) correctly point out that securely storing and using application-to-application credentials is essential in modern-day distributed applications. They further state that the use of passwords is convenient and quite flexible but extremely hard if not impossible to secure. Kostiainen, Ekberg, Asokan, and Rantala also point out that the use of hardware tokens as credentials are quite secure, but expensive and not practical. They propose an architecture for secure credential management called On-Board Credentials (ObC) which brokers connections between applications. Unfortunately, this system requires an isolated and secure execution environment for it to be effective. Once provisioned the requesting application is sent a connection key that is then used to connect to the remote application. The key is subject to misuse in the same manner as that of a password returned from a password vault. It would seem that the success of their methodology is keyed to the level of isolation the processing environment is able to achieve. The authors point out that additional analysis and testing is required to determine the level of security and usability of their proposal.

Blackwell (2009) states that insider attacks are on the rise and are very difficult to address in large part because insiders tend to possess high levels of privileged access rights and in-depth knowledge of system weaknesses. They propose a three-pronged systematic approach to mitigating the risk. The first is to reduce the attack surface, the second is to minimize impact zone and the third is to reduce the attacker's motivation either through persuasion or deterrence. They believe that by bolstering the worker's morale, increasing their pay and generally reducing the cost benefit of attacking their employer the frequency and scope of the attacks can be reduced. Considering that the United States has 5% of the global population and 25% of the world's prison population (Salerno, 2009) it would seem that Blackwell's hypothesis is flawed. The criminal generally does not consider the penalty during the commission of the crime. In fact the exact opposite holds true as they do not think they'll be caught at all. Blackwell is correct in his thinking that the best way to address this problem is to prevent the crime, although a different methodology is called for.

As the popularity of centralized disk storage systems grows so does the risk of unauthorized access to the files and data stored on those disks. These storage systems have become critical components of most corporate computing environments. Along with an increase in popularity and usage comes an increase in vulnerability and the risk of data loss and its damaging impact (Kher & Kim, 2005). Therefore, one must additionally consider the risk of an insider attack on centrally stored interpretive scripts which contain application-to-application credentials. A rogue system administrator with legitimate privileged access could perpetuate an attack on these scripts to harvest credentials simply by mounting a file system remotely from any number of servers that may or may not be adequately secured.

Chen, et. al. address the risk of file corruption and/or deletion through insider attacks on centralized storage systems and propose a secure and efficient Remote Data Checking (RDC) scheme for network coding-based distributed storage as a means to catch and correct inappropriate modification to files stored on central storage systems. In their research the central storage server is considered untrusted and their efforts are directed at preventing the destruction of files stored on the server by making those files recoverable. Their solution does not address access by authorized administrators for the purpose of credential harvesting. It does, however, illustrate the need to secure those files against attack (Chen, Curtmola, Ateniese, & Burns, 2010).

Kernel level security policy enforcement is the most widely used security architecture in systems to date. It is based on a two-tiered privilege architecture that allows for a single administrative super-user and all other users with no particular special privileges. These non-super-users are largely restricted to accessing files owned by themselves or groups in which they are members. The elevated privilege of the super-user is generally required to perform almost all administrative tasks. For this reason super-user accounts are highly exposed, difficult to defend and a favorite target of adversaries (Payne, 2007).

Payne (2007) acknowledges that cryptographic algorithms are in wide use enforcing networking security policies. He explains that passive cryptographic file systems exist primarily to protect the confidentiality of data stored on those files systems. Payne states that these systems break down when the super-user credentials have been exploited and the cryptographic keys obtained by attackers after compromising the privileged user's ID. It is surprising that with 17% of all insider attacks coming from the administrators themselves (Randazzo et al., 2005) Payne did not consider rogue administrators more in his thinking.

Payne (2007) proposes a system called Vaults which boasts of an enhanced access control system that encrypts the file system at the kernel level. This system can prevent a super-user from reading a file that is owned by a non-privileged user. This system works well in that it mitigates the risk of privileged users gaining access to critical files stored on the files system. Payne also claims that once the kernel is fully booted it will assume responsibility for verifying the integrity of trusted interpreted scripts against modification by a privileged attacker.

Payne's proposed kernel level key management architecture does provide a cryptography based security model that goes a long way to protect user files from access by privileged accounts. However, two major concerns weaken the value of the proposed solution. The first is that Payne does not consider that trusted code contained in interpreted scripts may have been compromised prior to being installed on the system and secured by Vaults. Unless the code is meticulously reviewed prior to rollout, it is quite easy for a rogue programmer to plant a logic bomb in the application that will capture the credentials once it is installed. Additionally, quite often system administrators are tasked with performing the software installs. It is not hard for a rogue administrator to insert malicious code into a script during the installation process. Programs with logic bombs inserted into their code can easily circumvent Payne's secure environment as Vaults only monitors changes made to trusted scripts after installation.

The second, and probably most important, concern is that anytime you modify the kernel of an operating system you risk voiding the manufacturer support of the system and software. In some cases modifications such as that which Payne proposes may even void the warranty. You also run the risk of negatively affecting the reliability of the data processing computations and results. If two plus two suddenly equals five, was it the floating point co-processor (FPU) in the central processing unit (CPU) that went bad or was it the kernel modification that caused the problem? In a highly critical computing environment standardized software installs are of the utmost importance. Anything that modifies the kernel is generally frowned upon and not allowed in most IT organizations. In order to be acceptable to communities performing critical processing, his proposal would have to be incorporated into the kernel, certified and supported by the manufacturer. Then, unless accepted by all manufacturers and made universally available across all platforms, the interpretive scripts accessing the Vaults environment lose their platform independence and in turn their portability.

Kher and Kim tell us that the confidentiality and integrity of centrally-stored data at rest can be achieved through cryptographic operations on the user side. This encryption requires the use of keys provided to the user for access to the storage system. In the case of Networked File Systems (NFS), which is widely used in the UNIX environment, they readily admit that it is the system administrators that control and issue user credentials for access to the shared storage. Therefore, the administrator who issues the user's credentials also has the ability to access their files masquerading as the user. Because NFS treats security as an afterthought it is ever more important to avoid storing credentials in scripts and to secure the handling of the credentials requested by an interpretive language script and returned from a secure credential vault (Kher & Kim, 2005).

Two areas of concern that are easily addressed, but often are ignored are the securing of backup media such as tapes and other portable devices often stored off-site for disaster recovery purposes and the disposal of storage devices that are no longer needed (Fendler, 2004). You can have the greatest and most secure file system in the world to protect interpretive language scripts with hard-coded application-to-application credentials, but it will all be for naught if someone were to harvest the credentials from a backup tape. This is easily addressed by encrypting backup tapes. However, only the newer tape devices such as LTO4, LTO5 and LTO6 (Linear Tape-Open) tape drives support hardware encryption ("Encryption Technology for HP StorageWorks LTO Ultrium Tape Drives," 2010). Software encryption is often slow, cumbersome and not practical in larger installations. Tape backup encryption is expensive and is generally not an option available to smaller less well-to-do organizations. Although encryption of tape backups is highly recommended it is best to not have the credentials hard-coded in the scripts that are stored on the backup media.

The second concern is the disposal of storage devices that are no longer needed. This is more a matter of policy than a scientific concern. All discarded storage devices should be wiped clean with a fairly inexpensive NSA certified degaussing device. Secure disposal can also be accomplished by hiring a shredding company who will certify the destruction of the media. Again, major regulated companies adhere closely to policies that require this. It is the smaller, less prominent organizations or organizations that are not regulated or audited that are either not aware of or cannot afford to enact this policy. The risk could be greatly mitigated if the credentials were not hard-coded into the software to begin with.

The practice of storing application credentials in cleartext in the application code, although frowned upon by standards bodies and internal auditing guidelines, has long been widely accepted as an unavoidable risk of doing business. It is deemed to be fully acceptable and is a common practice among IT professionals world-wide (Chumash & Yao, 2009). Chumash and Yao (2009) tell us that there is no framework available today that protects sensitive information from insider attacks and allows for the safe execution of interpretive scripts. Furthermore, they tell us that due to the interpretive nature of the scripts it is <u>impossible</u> to protect sensitive information contained in the scripts such as application-to-application connection credentials. Their approach to addressing this risk is one of reducing the impact of such insider attacks by focusing on early detection of data loss. While it is always prudent to adopt the measures suggested by Chumash and

Yao, it is rather too resigned to defeat to state as emphatically as they have that it is impossible to reduce the risk further than that which is known and understood today.

In part, accepting that application-to-application credentials cannot be protected stems from a lack of conceptual and practical security training for software developers ("Making Security a Business Priority," 2008; Said et al., 2009). Acceptance is also encouraged due to a void of functional alternatives coupled with the over-reliance on segregation of duties (SoD) as the primary risk mitigation methodology (Lieberman, 2010; Singleton, 2002). In a broad generalized sense, SoD is the concept of having more than one person required to complete a task. It is alternatively called separation of duties and specifically within the context of IT operations it is defined as separating the development community from the production processing environment and limiting access to just those who need access to support the production environment.

The International Organization for Standardization's (ISO) standard on information technology and security techniques, ISO/IEC 27001:2005, section 11.03.01, specifies that no hard-coded credentials should be allowed in any automated logon process, yet no insight is offered as to how one would go about securing an unattended batch process script (ISO, 2005). The Payment Card Industry's (PCI) Data Security Standard, section 8.5.16, accepts hard-coded passwords as an acceptable risk, but requires that the production environment protect the credentials against unauthorized use (PCI, 2009). This is virtually impossible if one considers that production computer systems and associated database management systems require real-time maintenance and support from system, application and database administrators who have the privileged access necessary to carry out their duties and also view the production script source code.

It is a significant when one considers the absence of warnings and the lack of research surrounding the hardcoding of credentials and/or the handling of those being returned from password vaults. In their 1,175-page definitive guide to software security assessments, Dowd, McDonald and Schuh (2007) made no mention of hard-coding credentials in software or the risk of exploitation of credentials when returned from a password vault. Another example of the absence of this issue in secure programming guidelines can be found in the Guimaraes, Murray and Austin (2007) paper on developing secure database programming courseware. Not a single mention of hard-coded credentials or the risk of exploiting credentials returned from a password vault can be found. George and Valeva (2006) identify the lack of application-to-application security and defensive programming techniques in undergraduate curriculum as a root cause for it being ignored in professional software development shops. This is further evident in Yang's paper (2009) suggesting methods for teaching database security and auditing which present the entire realm of application security in terms of SQL injection attacks and never once mention access control or the protection of connection credentials as a cross-cutting concern.

In their paper on database application security Said, Guimaraes, Maamar, and Jololian (2009) state that a lack of academic literature on database application security led them to develop courseware to educate developers, yet they still fail to mention the hard-coding of DBMS access credentials as a real security risk. Because academia has not recognized the protection of application-to-application credentials in interpretive scripts as a serious security risk, undergraduate and graduate courseware is sorely lacking in this area (Ge & Zdonik, 2007). It is not hard to understand why graduating students who become researchers tend to ignore areas of research in which they have been given no educational foundation. Furthermore, those graduating students who go on to careers in software development do not consider the protection of credentials in the design of their programs.

The Information Technology General Controls (ITGC), in section 404 of the Sarbanes-Oxley Act of 2002 (SOX), requires restricted access and SoD to reduce the risk of fraud through unauthorized data manipulation ("Sarbanes-Oxley section 404: A Guide for Management by Internal Controls Practitioners," 2008). SoD has long been thought of as the cornerstone of data protection (Mattsson, 2008). Mattsson suggests that the only way to address this risk is to combine SoD with data encryption at the database source. Encrypting the data and separating the security administration function from that of the developer/administrator thereby protects the sensitive data from database administrator attack. However, he later admits that even with these precautions the data is at risk to an attack from rogue administrators and developers who insert malicious code into the interpretive scripts and capture the returned data after its decryption.

In the context of this discussion, segregation of duties means that the development community is segregated from the production processing environment (Adaikkappan, 2009). This is additionally supported by CoBiT (Control Objectives for Information and Related Technology), a standard published by ISACA (Information Systems Audit and Control Association), which has achieved pervasive usage as a guideline for SOX section 404 compliance. Specifically, CoBiT sections AI 3.4 and AI 7.4 require the separation of developers from the production environment (ISACA, 2007).

SoD is the number one means for prevention of fraud and abuse in computer and application security. It is widely adopted in business, industry and government, and is the primary remediation focus in almost all audit findings (Gligor, 1998; Jianfeng, 2009; Mattsson, 2008). One area where the SoD process breaks down is in the nature of modern software development. In the past almost all computer programs were written in a language that required compiling before execution to convert program source code from a human-readable form to binary code that is readable and understood by the computer's operating system. Because binary program code is not human-readable and not easily modified, it fits well within the SoD scenario. Most software today is written in some form of interpretive scripting language such as Shell, Perl, JavaScript or BeanShell. These programs, called scripts, are human-readable and are essentially compiled at execution time. Application connection credentials are often stored within the script's code lines, thus opening up critical application access credentials for viewing by anyone who has read-access to the scripts (Lieberman, 2010). Gligor found that even with strong encryption policies in place and a mature access control framework implemented, it is virtually impossible to prevent the insertion of malicious code into an application that would allow unauthorized persons access to sensitive data. He concludes that to be effective, new administrative methods and stronger tools need to be developed that offer significant support for SoD policies (Gligor, 1998).

SoD effectiveness relies heavily on the detection of the fraud itself and is only as effective as the incentive program that encourages employee fraud reporting and the detection of fraudulent acts through follow-up audits. SoD's level of effectiveness as a preventive measure is high in areas of general application usage. It is less than effective when dealing with persons who have high level access rights or super-user privileges. Although implementing SoD in the overall application security framework is a sound practice, it does little to address the risks and issues arising from modern development and application support practices. Unfortunately it is most commonly relied upon as the end-all solution for protecting application-to-application credentials.

Jerbi, Hadar, Gates and Grebenev (2008) note the impact of insider attacks on a company's compliance with federal regulatory laws such as SOX and the Health Insurance Portability and Accountability Act (HIPAA). To address this concern they introduce the concept of least privilege access as a way to control administrator and application support personnel access to critical files, scripts and programs. Although this concept does restrict access by unauthorized administrators, Jerbi, Hadar, Gates and Grebenev's solution does nothing more than audit access by authorized administrators and application support personnel to scripts and programs containing hard-coded credentials. In addition, their solution does not address malicious code embedded in scripts and programs prior to the migration of the software from development to production.

Not securing hard-coded passwords stored in cleartext in scripts and programs as they migrate from development to production places a company's production systems at risk. Databases and other data stores become vulnerable to access by any and every developer that has had access to source code residing on development systems whether production system access is segregated or not (Woodbury, 2005). This seemingly puts the company in severe jeopardy of violating SOX and HIPAA requirements. Hicks, Rueda, St.Clair, Jaeger and McDaniel (2007) expand on the least privilege access model by incorporating a multi-level security framework to control and restrict credential use. Their proposal is built upon three security models: the Type Enforcement (TE) model, the Role-Based Access Control (RBAC) model and the Multi-Level Security (MLS) model. This framework, based on the Mandatory Access Control theory, requires that all subjects and objects be identified and marked. All security-sensitive operations are checked at runtime against a security policy to determine whether the operation should be allowed. This is again a sound policy; however, it does not address whether or not a file accessed by a qualified system administrator is for legitimate or illegitimate purposes.

Auditors have long cautioned against hard-coding credentials in scripts, but never seem to offer viable alternatives beyond separation of duties. The CPA Journal article on securing software (Rechtman, 2009) relies heavily on code reviews as the only sure way to secure software against exploits and misuse by closing security gaps before deployment. The author strongly discourages hard-coding critical values into the scripts and programs, but totally misses the risk of hard-coding credentials. The article also ignores the risk of software tampering after deployment. Martin points out that SoD weakens significantly unless used in conjunction with a strong Security Information and Event Management system (SIEM) to track and capture suspicious system activity (A. Martin, 2008). A truly efficient SEIM requires a hardened OS, such as Trusted Solaris, Trusted BSD or Security Enhanced Linux (SELinux), which is not economically practical or supportable in the average batch processing environment.

In their consideration and acceptance of the cleartext hard-coding of application credentials many auditing agencies and standards bodies recommend auditing the use of the credentials themselves as a method to mitigate the risk presented by this practice. For instance the audit guidelines published by the Institute of Internal Auditing (IIA) accepts that application-to-application connection credentials held within the software are necessary and recommends that the use of those credentials be monitored for possible misuse (Bresz, Renshaw, Rozek, & White, 2007). This seems to be a quite reactionary approach that is solely dependent upon the depth and chronological sensitivity of the audit log monitoring that takes place in the targeted firm. To be truly effective the monitoring must be real-time and intelligent. The only way to provide real-time intelligent monitoring is via a fully-automated log-scanning tool that can correlate activities across all monitored systems. These types of monitoring systems are complex, expensive and require an entire staff to maintain. In most cases they are prohibitive to all but the largest operations. Anything other than real-time monitoring is at best an *after-the-fact* approach that relies on the firm's security personnel to intercept the attacker before the data leaves the premises or before any malicious damage reaches the point of non-repair. It is a hit-or-miss approach at best.

There are numerous commercial products available that attempt to address the cross-cutting concerns of secure enterprise credential management. All products reviewed claim to secure application-to-application connection credentials used within programs and interpretive scripts. These products are generally referred to as secure password repositories or password vault software products. Although their feature sets vary greatly, they all have the same essential methodology. The products store enterprise passwords in a secure client-server back-end database with a front-end password broker process through which remote agents can request and retrieve a password from the vault. Communication between broker and remote agent is generally encrypted with agents caching requested passwords locally thereby reducing the load on the broker. Applications requesting a password are authenticated by the agent prior to passing the request onto the broker. There are varying degrees and methods of application-authentication used to verify the identity of the requesting application. These authentication schemes differ with each vendor's product, but all provide a similar method to vet the application prior to returning the requested credentials.

The e-DMZ' Password Auto Repository (PAR) technical product overview ("Application Password Management Module," 2009) states:

e-DMZ Security's Application Password Management (APM), part of the TPAM Suite of privileged user and access control solutions, provides a solution to replace embedded passwords that are hard-coded in scripts, procedures and programs with simple CLI/API calls. Often overlooked, embedded passwords create back-door access accounts to target systems and applications that can easily be exploited. Replacing these hard-coded passwords with programmatic calls that dynamically retrieve the account credential removes this often overlooked exposure.

An analysis of the company's product literature and a provided white paper ("Managing Embedded Application Passwords with Password Auto Repository™ (PAR)," 2009) shows that the product offering protects the password up to the point where it is returned to the requesting application. The product has no facility available to secure and protect the password after it is returned to the application. Although this is a step in the right direction it falls way short of addressing the risk presented by rogue programmers and administrators who could embed logic bombs in the code to harvest the returned password.

The Password Manager Pro (PMP) User Guide ("Password Management API for Application-to-Application Password Management," 2009) makes the following claim: “Any application or script can query PMP and retrieve passwords to connect with other applications or databases, eliminating hard-coded passwords.” No facility is provided to address concerns with handling the password after it is returned to the application. This product does include a facility to change the password after each use, which the company touts as a method for securing the password it returns. However, this falls down when you consider that the password could be hijacked and exploited between the time the password is returned from the vault and the time the password is changed. When asked, the company’s technical representative admitted that there was no known solution to addressing the insider threat posed by compromised code exploiting the returned password.

The Cyber-Ark Application Identity Management (AIM) implementation guide ("Application Identity Management Implementation Guide," 2009) states:

> Application credentials are often stored in embedded form in the application code, or in a configuration file, usually in cleartext that is visible to a large audience. This challenge identifies a security gap and significant risk, often captured by auditors, where these sensitive database and application ID passwords are widely known and accessible to developers, help desk engineers, etc.

This product boasts a patented password vault, a robust authentication process for password requestors and a highly-available, locally cached agent-based architecture that affords a well thought-out password protection scheme. The company further claims that by removing hard-coded passwords from application code and configuration files you can make “them invisible to developers and support staff.” When questioned, a Cyber-Ark technical support engineer reluctantly admitted that their product cannot address the insider threat posed by compromised code exploiting the returned password. They suggested that the passwords used by applications be changed every few hours as a means of mitigating this concern. However, a password can be compromised and an illicit connection be made to a data store within seconds of a legitimate password request being made to the vault and that password being returned to the application. This suggested solution can be deemed weak at best.

A white paper from Cloakware regarding their Password Authority (PA) password management software suite ("Cloakware Password Authority™," 2009) boasts that “Your unattended servers no longer need hard-coded credentials to access other servers.” It further boasts “It helps you meet your compliance requirements by eliminating shared and hard-coded passwords,…” Cloakware’s product offering has a secure password vault backend, a local agent that caches passwords for high availability and a large assortment of API’s by which applications can request and receive passwords.

Cloakware’s literature does not claim to solve the entire problem of password exploitation. In a question and answer session Cloakware’s Chief Technologist Robert Grapes admitted that their product only “...helps to prevent developers or administrators from having unmonitored access to production systems.” As with the other password vault technology products, Cloakware’s product offerings do not protect a legitimately requested and returned password from being captured and exploited once it is returned to the requesting application.

In 1999-2000 the Cloakware Corporation applied for and was granted a Canadian patent describing an invention in which a series of one-way hashed passwords are stored both in the remote resource and in the application requesting access. With each successive login attempt the accessing program sends the previous password in the series (Johnson, Gu, & Chow, 1999). This is a very unique approach in that it uses an unattended two-factor authentication scheme to authenticate the application requesting access. Both the resource and the requestor need to know what the current password is and what the previous one was. This prevents *man-in-the-middle* and *replay* attacks from capturing the password and reusing it. While, this solution protects the transmission of the credentials it does not protect the housing of the credentials in the program itself. Granted this patent award is almost ten years old, an eternity in the context of computer science, but it is worth examining to show that since the patent was awarded there has been no significant progress made toward addressing the issue of handling and protecting credentials in the application-to-application authentication process.

In their 2008 patent application Adams, Grapes, Gu, Mehan and Rong (2008), describe their invention as a method by which unattended software applications can request access to shared resources. This is noted in paragraph 0069, claim #5 commenting on claim #1, where they describe the process as returning resource credentials to the requesting application for use in establishing a connection to said resource. As well thought-out as this invention is it does not address issues concerning what the application does with the credentials after they are surrendered to the control of the application. As with the previously-explored solutions, the returned credentials are exposed to insider attacks by a rogue programmer who could embed password capture logic into his/her programs during the development cycle and now resides in the production processing environment.

Sade and Adar's (2008) patent application addresses the issue of insider attacks on credentials returned to requesting applications. The authors note that no one to date has effectively addressed the risk of exploitation from capturing a returned password and relaying it to an awaiting accomplice to use for illicit access to a remote data store. Sade and Adar propose a method by which the transmission of the credentials to the remote application or data store is intercepted and the password replaced with one extracted from a password vault, assuming the requesting application is fully vetted and authenticated to access the vault and the remote application/data store. Their patent application describes the invention as an interception module that performs intercepting methods such as application hooking, monitoring and intercepting network-packets, altering byte-code and altering operating-system drivers. They define application hooking as intercepting calls to APIs and/or modifying the API behavior.

The Sade and Adar (2008) solution, at first blush, seems quite attractive as it requires no modification to the requesting application code, does not use any hard-coded passwords placed in the code by the developers and is not subject to insider attacks in the form of embedded exploitive code placed in the application to trap and transmit the requested credentials. It is essentially a white hat, *man-in-the-middle* approach to solving the problem at hand. It seems that this approach would work well in situations where the network packets are not encrypted themselves or transmitted via SSL or TLS protocols. In addition, a firm understanding of each network protocol used by each individual vendor would be required to accomplish this. Intercepting the network packets also assumes the interception modules can overcome the deployed environment’s *man-in-the-middle* defenses.

The byte-code (binary) modification of system APIs and kernel driver code can be a slippery slope in which one can easily render an application, operating system or even the entire computer processing environment inoperable. Furthermore, tampering with APIs and kernel drivers, whether good-intentioned or not, introduces legal issues regarding software tampering and software licensing considerations. Lastly, the modification of the binary code for APIs and system kernel modules could possibly void associated support contracts and can cause concerns regarding system stability while running mission-critical batch processes. On the surface this seems like a great idea, but in the real world where down-time is costly, system stability is king and white hat hacking is frowned upon, this solution isn’t very palatable. It does, however, demonstrate the focus of present-day research and the relevance of protecting passwords from insider attacks.

Chapter 3

Methodology

Overview

The work presented here developed a method by which application-to-application connection credentials are better protected from various forms of attack. A framework was developed that adopts concepts learned from source code obfuscation research used in protecting programs from reverse engineering. Source code obfuscation is gaining an ever-growing level of importance in the secure software arena, affording both source code and binary protection in areas such as Intellectual Property (IP) and Digital Rights Management (DRM) (Giacobazzi, Jones, & Mastroeni, 2012). The goal of source code obfuscation is to transform and obscure the program variables and the codebase to such a point where it becomes unintelligible to both automated and human reverse engineering efforts (Majumdar, Drape, & Thomborson, 2007; Sosonkin, Naumovich, & Memon, 2003).

Reverse engineering efforts focus primarily on transforming binary objects into human readable source code which can then be used for illicit purposes such as copyright infringement or security bypassing. The problem addressed in this research focuses on code lines that are already in human readable form. Therefore, source code obfuscation in a pure sense is not applicable in this research. However, the basic principles and concepts derived from source code obfuscation research can be adapted and used as a foundation for this work. The framework established addresses the problem of securing application-to-application credentials such that an interpretive language script, one written in Perl for instance, can securely call a subroutine that establishes a connection to a data store or secondary application. During this process the credentials used to establish the connection is obfuscated, thereby reducing the risk of capture and exploitation.

A developmental research methodology was used to create a framework for application-to-application credential obfuscation. This framework was built upon the following concepts; the first being the disguising and obscuring of variables that hold the credentials being returned from a password vault. Obscuring storage variable names and breaking them up into multiple smaller variables will promote the obscurity of the variables and further reduce the risk of exploit.

Because Perl is a programming language that supports object-oriented programming, the second concept utilized an aspect in which the pointcut handled the password vault call, the reception of the returned password and the opening of the application connection. This takes place in the advice woven into the application connection open function call at runtime. The basis for this methodology is the obfuscation of the credentials handling to an area of program execution that is outside the program code accessible by the programmer and anyone else with access to the script code. This abstraction also increased the obscurity of the credentials to debugger attacks and memory scan attacks.

In preparation for developing the attack vector workload, and facilitating the development of the above framework, a batch processing environment similar to a typical small business was utilized. It consisted of a mix of computers running the three most prominent operating systems in use representing a typical batch processing environment.

They are Windows 7, Solaris 10 and Linux. All machines used in the research were networked together. A complete listing of hardware and software used in this research, including a network topography diagram, is available in Appendix B.

Attack Vector Taxonomy

Over the past three decades there have been numerous lists and taxonomies published to categorize vulnerabilities and attacks. Unfortunately, the degree of complexity and sophistication of current day attacks and vulnerability exploits renders these lists and taxonomies inadequate (Weber, Karger, & Paradkar, 2005). Within the context of this research the term *vulnerability* can be defined as a means whereby a hostile entity can successfully violate a system's security. We can also define an attack as the use of a tool or technique with which an attacker will attempt to detect and exploit a vulnerability to capture a password used in an interpretive script.

Hansman and Hunt (2005) published a paper reviewing the various taxonomies used to describe network and computer attacks. This paper reviewed early-published taxonomies including Bishop's 1995 vulnerability taxonomy, Howard's 1998 taxonomy, Lough's 2001 taxonomy and that which was published by the OASIS web application security council. Hansman and Hunt found that these taxonomies were too general in nature and weren't all capable of adequately classifying attacks. They proposed an alternative taxonomy for classifying attacks. To illustrate this taxonomy shortcoming one can look at the attack patterns classified in the CAPEC (Common Attack Pattern Enumeration and Classification) database. Of the 920 attack vectors classified under the Methods of Attack category only one sub-category deals with exploitation of privilege attacks and none address credential capture attacks ("CAPEC-1000: Mechanism of Attack," 2011).

The Hansman and Hunt taxonomy utilizes four dimensions for high-level attack classification; each of which can be broken down into levels or components which further granulizes the categorization. The first dimension categorizes the attack vector, the second dimension categorizes attack target, the third categorizes the vulnerability and the fourth categorizes the attack payload (Hansman & Hunt, 2005).

Their approach was attractive for use in this research as it allowed for attack categorization without specific external attack vectors or an attack vectors that are considered trivial. Attacks perpetrated by system administrators who already have legitimate access to the system tend to trivialize the attack vector. To illustrate the display of levels within a dimension, levels will be shown in the format of (*level* → *level 2* →*...level N*). *Password attacks* has been chosen as the first dimension's level one category and *exploiting implementation* as the level two category, thus the first dimension is displayed as (*Password attacks* → *exploiting implementation*).

The second dimension addresses the target of the attack. In this case the target is the interpretive script either containing the credentials or requesting them from a password vault. Therefore, (*Software* → *application* → *server* → *interpretive script*) seems appropriate.

The third dimension addresses the vulnerabilities the attack exploits. These are generally tied to vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE) database. However, the use of the CVE database is not appropriate for our purposes because CVE entries are specifically tied to vulnerabilities with published software programs, packages and applications. This research is focused on a programming concept and practice with scripts developed to demonstrate the concept and not a specific published application. Hansman and Hunt provide for such a case by allowing the use of Howard and Longstaff's (1998) *vulnerability in design* classification which defines the design of the program as perfectly implemented but flawed.

The fourth dimension addresses the attack payload which in this case is simply the capturing of the password. The Hansman and Hunt category selected is *Disclosure of Information*.

Dimension	**Level 1**	**Level 2**	**Level 3**	**Level 4**
One	*password attacks*	*exploiting implementation*		
Two	*software*	*application*	*server*	*interpretive script*
Three	*vulnerability in design*	*simple attack* *logic bomb attack* *debugger attack*		
Four	*Disclosure of Information*			

Table 1 Attack Classification Matrix

Attack Vectors Employed

The desired attack vector payload is to capture the credentials being used by the interpretive language scripts. This was accomplished using two attack vector methods and four attacks under those methods. The first method is a called a Man-At-The-End attack (MATE). This type of attack is generally defined as an adversary gaining an advantage by violating the software under their control (Collberg, 2011). Tampering attacks and reverse-engineering attacks are the two most common forms of MATE attacks (Falcarin, Collberg, Atallah, & Jabubowski, 2011).

For the purposes of this research the MATE defined tampering attack was used in the first attack vector method. The two MATE tampering attack vectors employed were a probe attack and a logic bomb attack. A probe attack is one in which an adversary with privileged access examines and probes software looking for low hanging fruit such as embedded credentials within the code of the script (Falcarin et al., 2011). There were two flavors of probe attack employed. One used simple commands to examine the scripts, the second used a debugger to examine the code and it's in memory variables during execution. A logic bomb attack is an attack in which the code of the script is modified to relay the credentials to an adversary at run-time. For this attack code was added to the script to pass the payload to standard output for capture.

The second attack vector method that was used is a memory scan attack. Researchers have had quite a bit of success in retrieving encryption keys from active system memory. Enck, Butler, Richardson, McDaniel and Smith (2008) found that in order for this attack vector to be successful the adversary must have physical privileged access to the machine on which the software is running as was the case in the research presented here. However, this research differs from Enck, Butler, Richardson, McDaniel and Smith's research in that the payload is not an encryption key, but a plaintext password. Hargraves and Chivers (2008) report that several researchers have had great success in retrieving plaintext passwords using the same attack vector. Bauer (2009) also reports on the success of rootkits in scanning memory and gaining access the privileged information.

The attacks that were used to evaluate the pre-and post-research exploitability of the test scripts are broken down below. Each attack was performed using a privileged account ('root' for Unix/Linux and 'administrator' for Windows). The privileged account and commands used varied depending of the operating system. The attacks were carried out against two scripts that performed a simple task; connect to an RDBMS and perform a simple query in a loop of 100 iterations and print the data returned. One script utilized hard-coded credentials and the second utilized credentials returned from a password vault.

(Vulnerability in design → simple attack)

Simple operating system-specific commands were used on each operating system (Solaris, Linux and Windows) to capture the password. Success of the attack was measured by the ability to capture the password and then by the degree of difficulty of a successful attack. This test was performed on a script that used a hard-coded password and a script that requested a password from a password vault.

The following commands were employed to extract the password from the scripts at rest:

Windows

- type – This command prints the contents of the script to standard output (the screen). The script code was examined looking for occurrences of password embedding or calls to a password vault. The examinations like this may yield additional areas to investigate such as other scripts that are called from the target script. Example command:

  C:\Users\Gary> type perlscript.pl
- find – This command searched the targeted script for occurrences of the specified token and printed them to standard output (the screen). Variations

on the word ‘password’ were searched for in the script. Example command:

```
C:\Users\Gary> find :password: C:\Users\Gary\perlscript.pl
```

Unix/Linux

- cat – This command prints the contents of the script to standard output (the screen). The script code was examined looking for occurrences of password embedding or calls to a password vault. The examinations like this may yield additional areas to investigate such as other scripts that are called from the target script. Example command:

  ```
  #: cat perlscript.pl
  ```

- grep – This command searched the targeted script for occurrences of the supplied token and prints them to standard output (the screen). Variations on the word ‘password’ were searched for in the script. Example command:

  ```
  #: grep –i password perlscript.pl
  ```

  In the above example the –i option makes the search case insensitive.

(Vulnerability in design → logic bomb attack)

Logic was embedded in the script to capture and distribute the password during the execution of the script. This test was performed on scripts that requested a password from a password vault. The exact nature of the code that was added to the script depended on the code structure of the script itself. Generally, passwords returned to a requesting script are stored in a localized variable for future use. The contents of that variable were written to standard output. Alternatively, the contents of the password variable can be sent to Standard Error or emailed out. However, those are really alternative methods of distribution after the fact. Therefore, writing the password to

Standard Out sufficed for this test. Success of the attack was measured by the ability to capture the password, and then secondarily by the degree of difficulty in achieving the successful attack.

The VI editor was used for this attack on all platforms. On the windows platform the MKS VI editor version 8.5, build 1397 was used. It is available from MKS, Inc. (www.mkssoftware.com). On the Linux and Solaris platform the VI editor included with the release of the O/S was used.

(Vulnerability in design → debugger attack)

A process debugger was used to step through the execution of the script in an attempt to capture the password during script execution. "Ptkdb is a free/open source debugger for Perl with graphical user interface (GUI) based on Perk/Tk." (Page & Marinov, 2007) Because it is specifically built for Perl, it is highly portable and able to run on Solaris, Linux and Windows 7. Being able to use the same debugger across platforms allowed for a more consistent platform independent test scenario.

This test used the Perl/Tk debugger, ptkdb version 1.231, to explore the script as it executed. The entire execution of the script was manually stepped through and all internal and external variables were examined looking for the possible storage of a password. At each step all sub-functions were stepped into and their variables examined. Drilling down continued until the lowest executing sub-function has been reached. Success of the attack was measured by the ability to capture the password and then by the degree of difficulty of a successful attack.

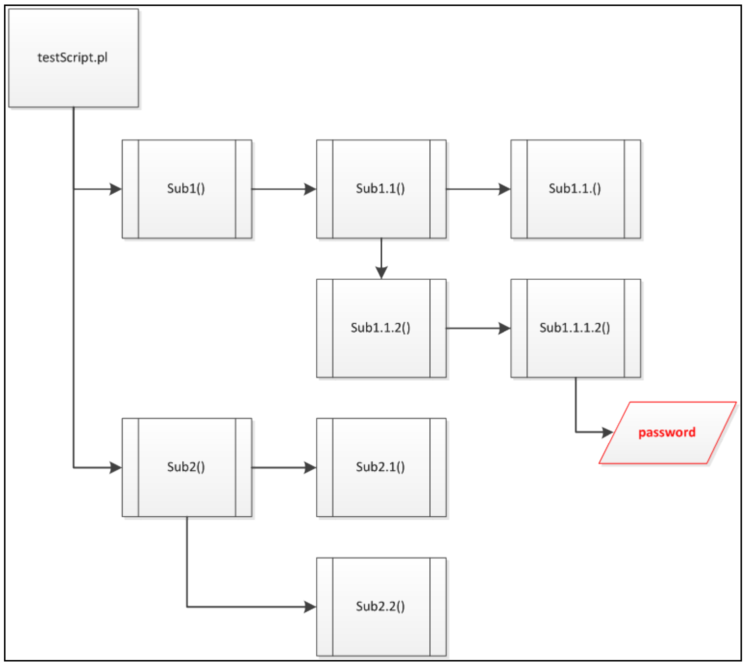
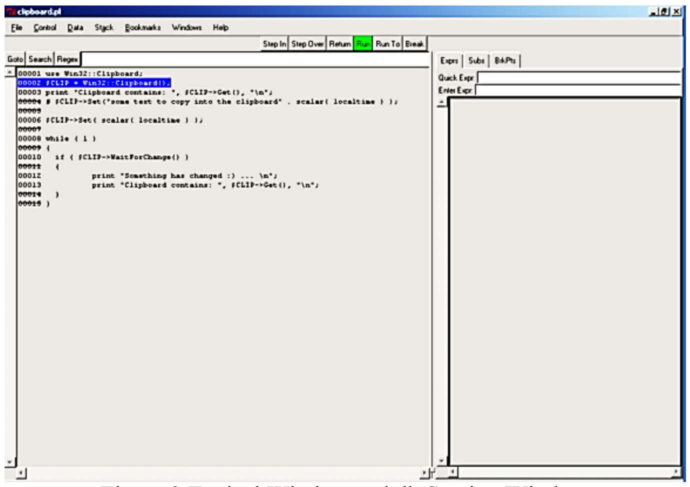


Figure 1 testScript.pl Function Exploration Example

Windows

- The script was executed using the following syntax:

C:\Strawberry\Perl\Bin> perl –d:ptkdb testScript.pl.


clipboard.pl
File Control Data Stack Bookmarks Windows Help
Step In Step Over Return Run Run To Break
Goto Search Regex
Exprs Subs BrkPts
Quick Expr
Enter Expr

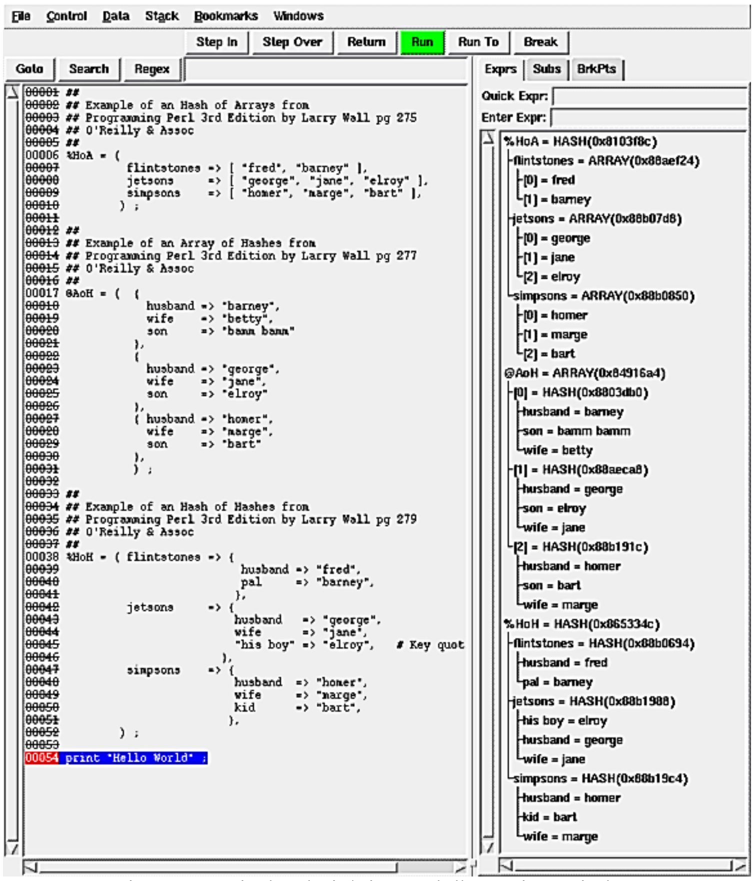
```
use Win32::Clipboard;
$CLIP = Win32::Clipboard();
print "Clipboard contains: ", $CLIP->Get(), "\n";
# $CLIP->Set("some text to copy into the clipboard" . scalar( localtime ) );

$CLIP->Set( scalar( localtime ) );

while ( 1 )
{
  if ( $CLIP->WaitForChange() )
  {
          print "Something has changed :) ... \n";
          print "Clipboard contains: ", $CLIP->Get(), "\n";
  }
}
```

Figure 2 Typical Windows ptkdb Session Window

Unix/Linux

- The script was executed using the following syntax:

#: perl –d:ptkdb testScript.pl.



```
##
## Example of an Hash of Arrays from
## Programming Perl 3rd Edition by Larry Wall pg 275
## O'Reilly & Assoc
##
%HoA = (
            flintstones => [ "fred", "barney" ],
            jetsons     => [ "george", "jane", "elroy" ],
            simpsons    => [ "homer", "marge", "bart" ],
          ) ;

##
## Example of an Array of Hashes from
## Programming Perl 3rd Edition by Larry Wall pg 277
## O'Reilly & Assoc
##
@AoH = (  {
               husband => "barney",
               wife    => "betty",
               son     => "bamm bamm"
             },
             {
               husband => "george",
               wife    => "jane",
               son     => "elroy"
             },
             { husband => "homer",
               wife    => "marge",
               son     => "bart"
             },
             ) ;

##
## Example of an Hash of Hashes from
## Programming Perl 3rd Edition by Larry Wall pg 279
## O'Reilly & Assoc
##
%HoH = ( flintstones => {
                           husband => "fred",
                           pal     => "barney",
                          },
           jetsons     => {
                          husband   => "george",
                          wife      => "jane",
                          "his boy" => "elroy",    # Key quot
                        },
           simpsons    => {
                          husband  => "homer",
                          wife     => "marge",
                          kid      => "bart",
                         },
          ) ;

print "Hello World" ;
```

```
%HoA = HASH(0x8103f8c)
├flintstones = ARRAY(0x88aef24)
│├[0] = fred
│└[1] = barney
├jetsons = ARRAY(0x88b07d8)
│├[0] = george
│├[1] = jane
│└[2] = elroy
└simpsons = ARRAY(0x88b0850)
 ├[0] = homer
 ├[1] = marge
 └[2] = bart
@AoH = ARRAY(0x84916a4)
├[0] = HASH(0x8803db0)
│├husband = barney
│├son = bamm bamm
│└wife = betty
├[1] = HASH(0x88aeca8)
│├husband = george
│├son = elroy
│└wife = jane
└[2] = HASH(0x88b191c)
 ├husband = homer
 ├son = bart
 └wife = marge
%HoH = HASH(0x865334c)
├flintstones = HASH(0x88b0694)
│├husband = fred
│└pal = barney
├jetsons = HASH(0x88b1988)
│├his boy = elroy
│├husband = george
│└wife = jane
└simpsons = HASH(0x88b19c4)
 ├husband = homer
 ├kid = bart
 └wife = marge
```

Figure 3 Typical Solaris/Linux ptkdb Session Window

(Vulnerability in design → memory dump attack)

This attack attempted to extract the password from a dump of system memory during script execution. Methods and commands used varied depending on the operating system the script was running on. Success of the attack was measured by the ability to capture the password and then by the degree of difficulty of the successful attack. The following steps were used for each operating system:

Windows

Several tools from different manufacturers were used in conjunction to generate and search a memory dump (see Appendix B). The following commands will be used to create and explore the Windows memory dump:

- As administrator, using the Microsoft Sysinternals LiveKD kernel debugger, a memory dump of the system's memory was generated during the execution of the targeted Perl Script. The following syntax was used:

  ```
  0: KD> .dump –f C:\memory.dmp
  ```

- The following is an example of a typical simple parsing of the memory dump searching for a 'Password' string:

  ```
  C:\> type memory.dmp | strings | grep –i 'password = '
  ```

Unix/Linux

The following commands were used to create and explore the Unix/Linux memory dump:

- As root, using the standard Unix/Linux dd command, a memory dump of the system's memory was generated during the execution of the targeted Perl Script. The following syntax will be used:

  ```
  # dd if=/dev/mem of=/memory.dmp
  ```

- The following is an example of typical command syntax for a simple parsing of a memory dump searching for a 'Password' string:

  ```
  # cat /memory.dmp | strings | grep –i 'password = '
  ```

Password Obfuscation Design Method

Because Aspect code can be housed outside of the targeted program and called transparently during execution a higher level of obscurity can be achieved, further reducing the risk of memory scan attacks. A higher level of obscurity, resulting in a lower level of risk, was achieved by implementing concerns that utilize the Composition Filter (CF) model. The CF model is a modular extension to the conventional objectbased model of which Perl subscribes. Filters define enhancements to messages sent and received by objects (Filman, Elrad, Clarke, & Aksit, 2005).

Consider a Perl script that calls a module that opens a connection to a remote data source such as an RDBMS. The module received messages containing information needed to affect the desired end result, which is an established connection to the remote data source. The credentials needed to authenticate the connection are generally passed to the called module from the calling script.

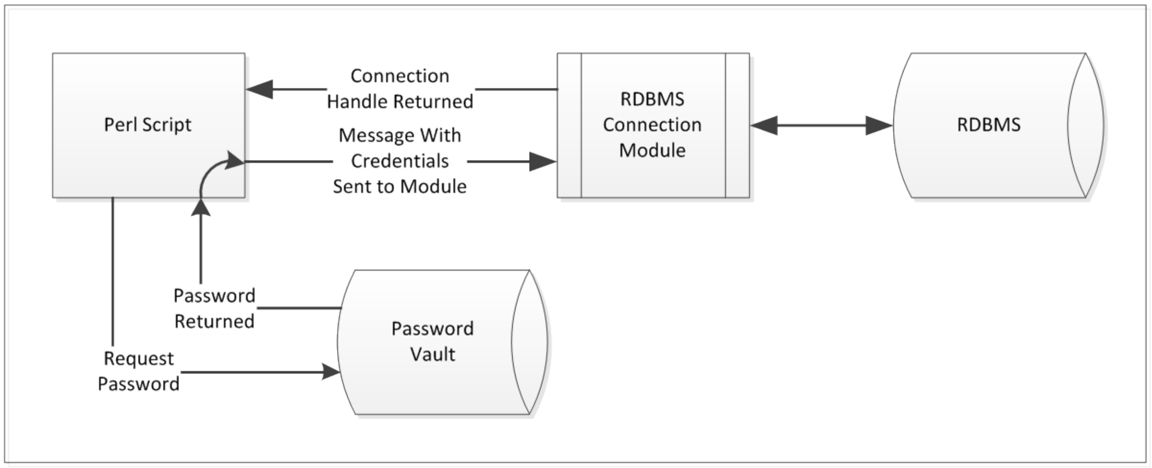


Figure 4 Typical Database Connection Scenario

Figure 5 illustrates the proposed filter and how the pointcut intercepts the connection credentials, makes the password vault request and subsequently modifies the message with the correct credentials before passing it on to the RDBMS connection module. By enhancing the message in this manner a level of abstraction is introduced that obfuscates the handling of the credentials. This is accomplished in part because the code for the aspect does not reside in the program code nor is it placed in the program codebase by the developers who wrote the program. The Aspect will be located in the function library and loaded at program execution time.

At no time are the credentials returned to the program or handled in any manner by the script code that was created by the program creator. Because there is no code in the script itself that touches the credentials the level of obfuscation is high and the risk of a successful attack is diminished greatly.

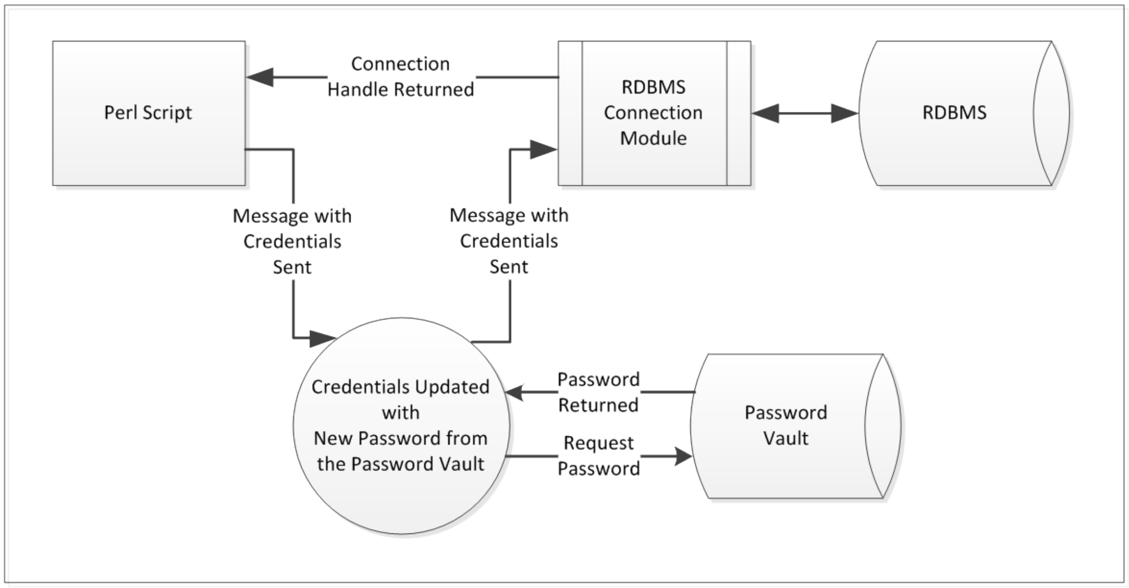

Perl Script
Connection Handle Returned
RDBMS Connection Module
RDBMS
Message with Credentials Sent
Message with Credentials Sent
Credentials Updated with New Password from the Password Vault
Password Returned
Request Password
Password Vault

Figure 5 Connection Process with Composition Filter Enhancement

Baseline Attack Vector Payload

The initial baseline demonstrates the exploitability of the credentials stored in the interactive scripts as well as those scripts that use the Cloakware Password Authority product. The security and strength of the Cloakware product was not evaluated, as it was used simply as a transport for returning the password to the application. These interpretive Perl scripts typify a common batch processing environment in which scripts run unattended, connect to a database server and perform standard database-related tasks such as selecting, inserting and deleting data from tables under database control. Typical database servers require password authentication from accessing scripts. Common to batch processing environments are a mix of computers with disparate operating systems. It is also significant to note that attack vectors vary with operating systems. Therefore, computers running Windows, Linux and Solaris were used to emulate a typical batch processing environment allowing a complete and robust test environment. Vulnerability testing was conducted on all platforms and operating systems listed above. All discovered vulnerabilities and exploits were fully documented and are reproducible.

The Perl scripting language was chosen as the programing language for this research. It is platform agnostic, and by far, the most popular interpretive scripting language in use today for batch processing and web scripting (Sheppard, 2000). Several interpretive Perl scripts were developed that log into a database server and perform queries against several tables containing test data. These were not elaborate or complex scripts, but ones that simply demonstrated a connection to a database server using an ID and password to authenticate in a manner most typical to client/server architecture based processing environments. A commercial password vault product was used to store and return passwords to these programs. The Password Authority product from Cloakware was used to house and return a password upon request. Cloakware, a subsidiary of the Xceedium, Inc. was petitioned and has graciously granted this author a one year software license for their product use in this research (see Appendix A). Cloakware agents were installed on all test machines to facilitate automated requesting and receipt of passwords.

Two scripts were used for the creation of baseline testing. Each script initialized a connection to a remote data source (RDBMS) and performed a simple database query. The first script had the connection credentials hardcoded into the script. The second script is a bit more complex in that it did not have the credentials hardcoded into it, but relied on requesting the credentials from the password vault. It then passed the credentials on to the connection module. See Appendix C for a listing of each script.

Quantifying Testing Results

As introduced in the “Dissertation Goal” section of Chapter 1, Kepner and Tregoe’s Weighted Decision Analysis Matrix theory was used to quantify the testing results and measure the delta between the baseline and post-research results. The required matrix consists of an array presenting pre-and post-research tests on the vertical axis. The horizontal axis represents a list of attacks that were performed against the test scripts attempting to mine the password used in the scripts. Each Attack Vector was weighted (W) from one to ten to indicate the degree of difficulty as evaluated against all other exploits attempted. The score or degree of success for each attack (S) was presented in the vector (xy coordinates) of the pre-or post-research test and the attack. The weighted score (W x S) was calculated as the product of the attack weight and the degree of success of the attack (the score). The rightmost column contains the sum of weighted scores for the pre-and post-research testing. The delta between the total weighted score of the pre-and post-test results indicates the success or failure the method in reducing the risk of password exploit.

Research Risk Analysis Matrix Aspect Modified		Probe Attack		Logic Bomb Attack		Debugger Attack		Memory Scan Attack		
		Weight (W):	0	Weight (W):	0	Weight (W):	0	Weight (W):	0	
		Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Risk Score
Windows 7	Simple Script	0	0	0	0	0	0	0	0	0
	Passwd Vault Script	0	0	0	0	0	0	0	0	0
Linux	Simple Script	0	0	0	0	0	0	0	0	0
	Passwd Vault Script	0	0	0	0	0	0	0	0	0
Solaris 10	Simple Script	0	0	0	0	0	0	0	0	0
	Passwd Vault Script	0	0	0	0	0	0	0	0	0
	Risk FactorTotals		**0**		**0**		**0**		**0**	**0**

Figure 6 Sample Risk Analysis Matrix

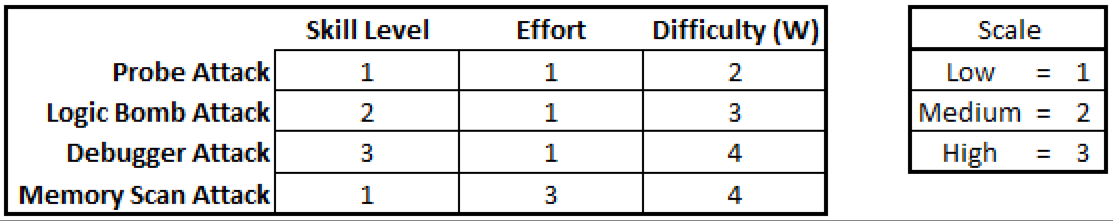
Evaluation of the attack success (the score) was first be based on whether or not the attempt to capture the password was in fact successful. A failed attack would automatically generate a score of zero. A successful attack is defined as the capture of the password. Once successful, an attack was then evaluated on the degree of difficulty in capturing the password. Each successful attack was evaluated on a scale of one to ten and then considered in determining the total score for that attack vector.

Assessing the degree of difficulty of an attack (W) is a process of evaluating the degree of technical knowledge and expertise necessary to carry out the attack along with the effort necessary. (Hongyu, Hu, Huang, Wang, & Chen, 2011) Byres, Franz, and Miller (2004) also point out that risk is an expression of the likelihood that a treat can exploit a payload, and that the likelihood of the attack success is directly related to the technical skill level of the attacker. They break down attack difficulty into four levels; *trivial, moderate, difficult* and *unlikely*.

Zaobin, Tang, Wu, and Varadharajan (2007) break down attack difficulty into a descending scale of five levels with level five being *very hard*, *hard, moderate, easy* and level one being *very easy*. The Verizon Risk Team ("2012 Data Breach Investigations Report," 2012) investigators define four levels of attack difficulty as *very low* – no special skills needed, *low* - basic skills needed, *moderate* – skilled techniques required and *high* – advanced skills required.

For the purposes of this research, attack difficulty (W) is defined as the required skill level of the attacker as influenced by the likelihood of the success of the attack. The likelihood of attack success is directly correlated to the effort required of the attacker to achieve the payload. For instance, a debugger attack requires a higher level of attacker skill than a memory scan attack. However, the memory scan attack requires a much greater effort to achieve success than the debugger attack. Because of the effort involved, the memory scan attack has a much lower return on investment than that of the debugger attack. Therefore, the debugger attack is deemed to have a lower degree of difficulty than the memory scan attack

Ranking and rating the difficulty of attack arguably involves a degree of subjectivity ("2012 Data Breach Investigations Report," 2012). Such was the case in determining the degree of attack difficulty for each of the four attacks considered in this research. The degree of difficulty was assessed on two planes. The first being the skill level required for the attack and second the effort involved in effecting the attack. Each plane was rated on a scale of one to three, whereas one was the lowest and three the highest ranking. Table two represents how the attack difficulty value for each attack was determined.



	Skill Level	Effort	Difficulty (W)
Probe Attack	1	1	2
Logic Bomb Attack	2	1	3
Debugger Attack	3	1	4
Memory Scan Attack	1	3	4

Scale		
Low	=	1
Medium	=	2
High	=	3

	Skill Level	Effort	Difficulty (W)
Probe Attack	low	low	10
Logic Bomb Attack	low	moderately low	9
Debugger Attack	moderate	moderately low	8
Memory Scan Attack	moderately low	very high	4

Table 2 Attack Difficulty Matrix

Resource Requirements

A small processing environment was setup that is representative of a typical client/server batch processing environment. A networked environment was assembled consisting of two commercial relational database managers (RDBMS), a commercial password vault and a mix of computers that represent the three major batch processing operating systems; Linux, Solaris and Windows. Xceedium's Cloakware Password

Authority was chosen for use in this research. Appendix A contains the formal request and permission to use the Cloakware product. Appendix B contains a detailed list of hardware and software that was used and a network topography diagram.

It should be noted that although not explicitly pertinent to the research, certain indirectly pertinent equipment is itemized in the list of resources. Equipment such as UPS power equipment, NAS storage devices and a tape backup server are germane to protecting the research environment and allowing for restoration of critical work should something untoward happen such as a power outage, hardware failure or an accidental deletion of critical files.

Chapter 4

Results

Introduction

In this chapter the results of the credential obfuscation research will be presented. The first part will describe the establishment of a risk level baseline against which the degree of success of the credential obfuscation method will be judged. The baseline is presented in the form of a set of risk analysis matrices as described in chapter three. Throughout the rest of this work the initial activity will be referred to as '*Baseline'* and the post baseline activity will be referred to as *'Aspect Modified'*. It is appropriately named because the code lines used to form the baseline scores were modified to utilize an aspect as the foundation of the credentials obfuscation method.

Attack vector baselines were established using two Perl scripts written specifically for this work and tested on three different platforms, Windows 7, Linux and Solaris 10. Each script established a connection to an RDBMS and performed a simple query in a loop of 100 iterations. In most cases it was not necessary to run the script through all 100 iterations to accomplish the targeted attack. Scripts running on Windows 7 and Linux connected to a MySQL RDBMS and scripts running on Solaris connected to an Oracle RDMBS. The only difference between scripts that were run on Windows and Linux and the scripts run on Solaris is name and IP address of the target RDBMS. Otherwise all scripts and Perl modules are identical. The brand of the target RDBMS was deemed insignificant to the work and research being performed and was chosen simply for ease of use. The method for connecting to the target RDBMS, the Perl DBI module, is germane to this work and was identical through all three platforms tested.

The first script utilized hard-coded credentials to connect to the RDBMS. The second script utilized credentials that were stored in a password vault and returned to the script via a subroutine called passwdLookup(). This subroutine wraps the password vault specific code into a single subroutine call. The passwdLookup() subroutine is contained in the Perl module called NOVA::Passwd.pm. It takes two arguments when called. The first is the ID for which the password is required and the second is the name of the Server the ID will be used to connect to. The two arguments are combined to form a single ID Alias and passed on to the password vault. The use of an ID alias is unique to the brand of password vault used for this research. Using the Perl module and the passwdLookup() subroutine allowed for the moving of redundant code from the test scripts to a library (Perl module). A full code listing of all scripts can be found in Appendix C. Wrapping the password vault product specific code in a Perl module is typical of the generally accepted method for using the Cloakware product from within a Perl script. It is also essential to obfuscating the password retrieval method and aided in the creation of the aspect code in the second part of the research.

Once baselines were established the Perl module was modified to include code that automatically invokes an aspect in which the pointcut handled the password vault call, the reception of the returned password and the opening of the application connection. This takes place in the advice woven into the application connection open function call at runtime. The exact same attacks were carried out against the scripts using the aspect modified Perl module. A second set of statistics was gathered and will be presented in the form of a second set of risk analysis matrices. Conclusions and analysis of success or failure will be drawn from the delta of the two sets of matrices.

During the execution of the attacks, it was observed that the level of attack difficulty stayed static between platforms and did not vary. The commands differed for each platform, but each attack vector, whether successful or not, met with an equal challenge on across platforms. This did not change between the establishment of the baseline scores and the establishment of the aspect modified scores.

Attack Vector Baselines

It should be noted that all successful baseline attack vectors were given the highest score of ten points. Attack vectors run against Aspect modified code were awarded scores based on the degree of difficulty in capturing the payload as compared to the baseline. The baseline Risk Matrix was identical across all three platforms.

Research Risk Analysis Matrix Baseline		Probe Attack		Logic Bomb Attack		Debugger Attack		Memory Scan Attack		
		Weight (W):	2	Weight (W):	3	Weight (W):	4	Weight (W):	4	
			(W x S)		(W x S)		(W x S)		(W x S)	
		Score (S)	Factor	Score (S)	Factor	Score (S)	Factor	Score (S)	Factor	Risk Score
Windows 7	Simple Script	10	20	10	30	10	40	10	40	130
	Passwd Vault Script	2	4	10	30	10	40	10	40	114
Linux	Simple Script	10	20	10	30	10	40	10	40	130
	Passwd Vault Script	2	4	10	30	10	40	10	40	114
Solaris 10	Simple Script	10	20	10	30	10	40	10	40	130
	Passwd Vault Script	2	4	10	30	10	40	10	40	114
	Risk FactorTotals		72		180		240		240	732

Table 3 Baseline Risk Analysis Matrix for all platforms

The probe attack vector was given a weight value of two as this is considered the easiest attack to carry out and required the least amount of skill. The logic bomb attack vector was given a weight of three as it is marginally more difficult to carry out than the probe attack vector and required some minimal programming skills. These two tampering attacks are considered most common forms of MATE attacks and therefore received the lowest weights (Falcarin et al., 2011).

Of the two remaining attack vectors the debugger attack vector and the memory scan attack vector were each weighted at four. Although the skill set required to effect the debugger attack vector is much greater than that which is required for the memory scan attack vector, it required less effort to carry out and is considered more common by MATE standards (Falcarin et al., 2011). The memory scan attack vector received a weight of four because it required much more effort to carry out than the debugger attack vector, but also required much less skill. It is quite time consuming and requires a brute-force effort to test strings found in memory to see if they are in fact the targeted payload being searched for. There is a very low return on investment for the effort in this kind of attack.

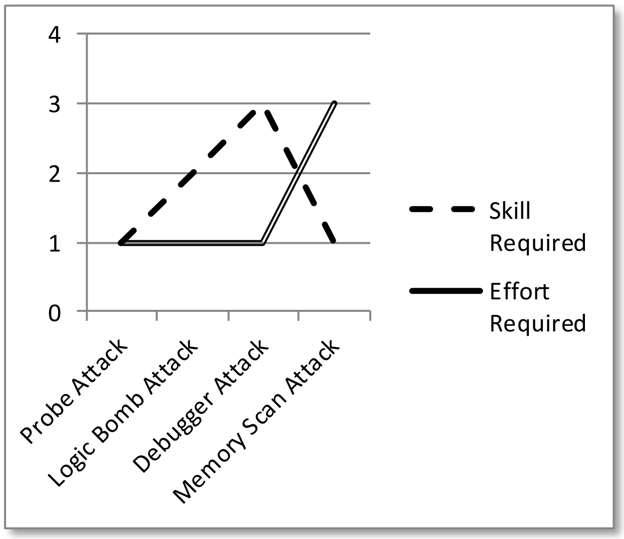


Table 4 Attack Vector Weight to Skill Level Comparison

Probe Attack Vector

The first attack performed was a simple probe attack (Vulnerability in design → simple attack). It consisted of printing the targeted script to standard output. To facilitate this, the '*type*' command was used against the script with the hard-coded credentials on the Windows 7 platform and the '*cat*' command was used on the Linux and Solaris 10 platforms. Screenshots of each exploit can be found in Appendix D. A baseline score of ten indicating a successful attack was awarded to this attack vector when run against the script with hard-coded credentials.

When the attack was run against the script utilizing the password vault for password storage no credentials were able to be captured. However, the method for retrieving the password from the vault was easily discovered by the attacker during the attack. Even though capturing the payload was not achieved, valuable information was gained that could aide attackers in developing an alternative plan to capture the payload. A score of two was awarded for this attack because of the information gained during the attack. There was no variance between platforms for ease of execution or degree of difficulty in achieving success.

Logic Bomb Attack Vector

The second attack vector consisted of a logic bomb attack (Vulnerability in design → logic bomb attack). Code was inserted into the script to print the credentials to standard output. Screenshots of each exploit can be found in Appendix D. A score of ten was awarded this attack vector when run against both of the scripts as capturing the payload was easily accomplished. There was no variance between platforms for ease of execution or degree of difficulty in achieving success.

Debugger Attack Vector

The third attack vector (Vulnerability in design → debugger attack) utilized a debugger to capture the payload. When run against the script with the hard-coded credentials the total time of attack was less than ten seconds as the password was displayed in the initial debugger screen without having to execution any commands. When run against the script that utilized the password vault a total time to payload capture was approximately 2 minutes average across all platforms and required a higher degree of skill than the probe and logic bomb attack vectors. Screenshots of each exploit can be found in Appendix D. Simple debugger steps where used to accomplish the goal. The script was executed and stepped though one line of code at a time. It was not necessary to step into any subroutines, declare any break points or use any complex debugger skills to achieve success.

Memory Scan Attack Vector

The fourth attack vector utilized a memory dump and subsequent scan to capture the payload. It was observed that previous execution of the scripts seem to stay resident in the memory at the time of the dump. This held true even after cold system reboots. In order to accurately test attack vectors it was necessary to change the password used fairly often. A second observation was that the memory dumps for the Windows 7 platform and the Linux platform where identical in structure, while the memory dump for the Solaris 10 platform was quite different. The similarity of the Windows 7 and Linux memory dumps is most likely due to their both running same x86 based instruction set CPU architecture. The Windows 7 machine ran on an Intel 2.76 GHz i7 CPU M620 and the Linux platform ran on a 2.10 GHz AMD Athlon 64 X2 4000 CPU. The Solaris 10 platform ran on a 1.34 GHz UltraSPARC IIIi CPU which is an entirely different chip instruction set CPU architecture.

Searches for the string '$Passwd =' on the Windows 7 and Linux memory dumps yielded immediate success when attacking script with hard-coded credentials. When the same attack vector was run on the Solaris 10 platform the password was captured, but there were no easily identifiable characteristics associated with the password. Therefore, on the Solaris 10 platform a lengthy brute-force method of testing strings to identify which one was the password would have had to have been performed. This was similar in manner to that which was necessary for attack vectors on all three platforms against the script utilizing the password vault.

To put the brute-force identification method into proper perspective, the Windows 7 platform had ~20 million strings to parse, the Linux platform had ~28 million strings to parse and the Solaris 10 platform had ~5 million strings to parse. Curiously, all three platforms were configured for four GB of usable memory, yet the number of strings varied quite a bit. This made it much more difficult to capture the payload when the attack vector was run against the script utilizing the password vault. For the purposes of this research and because the password was known ahead of time, searches were performed looking for the known password and the brute-force method of identification was avoided. This had no negative affect on the data collected nor did it affect the outcome of the testing. Screenshots of memory scan attack vectors can be found in Appendix D. This level of difficulty only supports the lower attack weight assigned to this attack vector. In the wild only the most skilled and patient attackers would be able to carry out such an undetected attack successfully.

Findings

To achieve the goal of this research several modifications were made to the test scripts and the associated Perl module used to establish the baseline statistics. The first modification was to add the ‘use NOVA::Passwd;’ pragma to load the Perl module into the script with the hard-coded credentials when it was executed. The initial thought was to remove the hard-coded credentials completely from the script after adding the pragma call. However, the thinking changed during the testing of the script with the aspect firing.

A set of invalid hard-coded credentials were left in the script that served as an additional form of obfuscation, thus further masking the fact that an aspect is firing.

Modifications were also made to the second script that used the password vault for credential management. The password vault call code was removed and a set of hard-coded credentials were added as a smokescreen. When finished, both scripts were identical.

Further changes were made to the Perl module. A subroutine was added that fires off an aspect creating a filter. The pointcut intercepted the connection credentials passed to the RDBMS connection call and made the appropriate password vault request subsequently modifying the message with the correct credentials before passing it on to the RDBMS connection module. By enhancing the message in this manner a level of abstraction was successfully introduced that obfuscated the handling of the credentials.

The following table shows an analysis of the attack vectors run against scripts using the aspect modified Perl module. Each attack vector will be presented and analyzed in the following sections. A comparison of the baseline and aspect modified results will be presented in the summary section of this chapter.

Research Risk Analysis Matrix Aspect Modified		Probe Attack		Logic Bomb Attack		Debugger Attack		Memory Scan Attack		
		Weight (W):	2	Weight (W):	3	Weight (W):	4	Weight (W):	4	
		Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Score (S)	(W x S) Factor	Risk Score
Windows 7	Simple Script	0	0	0	0	5	20	10	40	60
	Passwd Vault Script	0	0	0	0	5	20	10	40	60
Linux	Simple Script	0	0	0	0	5	20	10	40	60
	Passwd Vault Script	0	0	0	0	5	20	10	40	60
Solaris 10	Simple Script	0	0	0	0	5	20	10	40	60
	Passwd Vault Script	0	0	0	0	5	20	10	40	60
	Risk FactorTotals		**0**		**0**		**120**		**240**	**360**

Table 5 Aspect Modified Risk Analysis Matrix for all platforms

Probe Attack Vector

The probe attack vector presented the largest variances between baseline and aspect modified scores. By utilizing an aspect to intercept the call to the DBI->connect subroutine and replace the supplied password with the real password the level of risk was reduced 100%. All attempts to scan the script code to capture the password were unsuccessful.

Simply converting the script to use the password vault thwarted the probe attack from capturing the payload. However, using the aspect to retrieve the password allowed the script to present the setting of a fictitious password as a smokescreen and pass it as an argument to the DBI->connect subroutine. This also further obscured the method of password retrieval from the attacker.

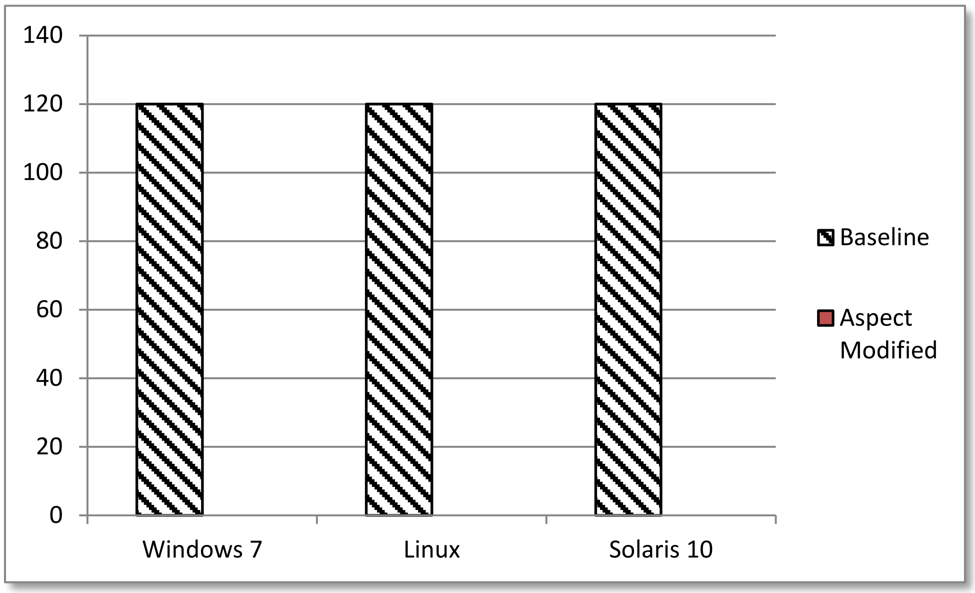


Table 6 Probe Attack Risk Reduction Comparison

Logic Bomb Attack Vector

The logic bomb attack vector presented the largest variance between baseline and aspect modified scores. By utilizing an aspect to intercept the call to the DBI->connect subroutine and replace the supplied password with the real password the level of risk was reduced 100%. This attack vector modifies the script code to print out the hard-coded password, or in the case of the script retrieving the password from the password vault the returned password from the vault. Only a simple edit on the script with the hard-coded credentials to add the ‘use NOVA::Passwd;’ pragma was required to reduce the risk of attack by 100%.

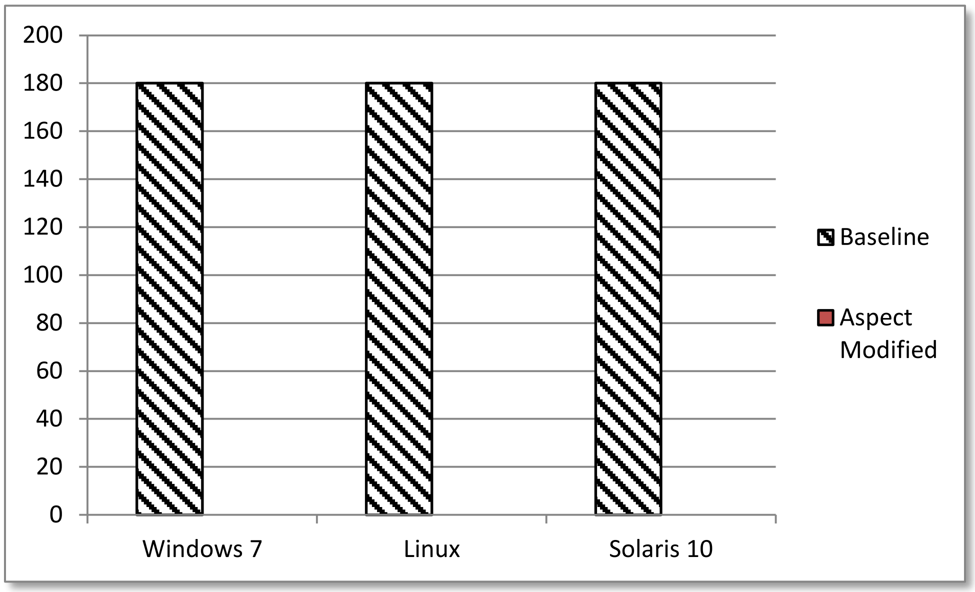


Table 7 Logic Bomb Attack Risk Reduction Comparison

In addition to adding the ‘use NOVA::Passwd;’ pragma to the script that utilized the password vault, the code that made the password vault call was removed. In its place the hard-coding of a password was added. This gave the script a higher level obscurity masking the use of aspect to retrieve the credentials.

Debugger Attack Vector

The debugger attack vector successfully captured the payload in both the baseline tests and the aspect modified tests. The level of difficulty and the skill required increased significantly when an aspect was introduced. On average 50 plus debugger commands were needed to capture the payload when calling the aspect to return the password from the vault. In comparison, no commands were needed for the hard-coded credentials script in the baseline test and three commands for the password vault baseline script.

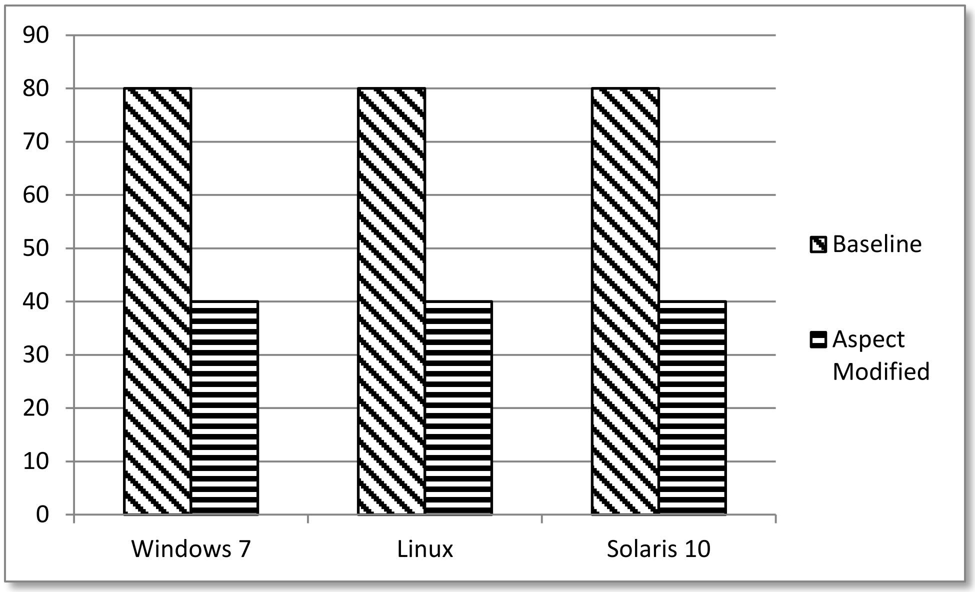


Table 8 Debugger Attack Risk Reduction Comparison

During the execution of the 50 plus debugger commands hundreds of lines of code required review and numerous subroutines had to be stepped into before the aspect code was discovered. It was clear that prior knowledge of the code base aided the attack and mostly certainly reduced the steps involved in achieving success.

Memory Scan Attack Vector

Both the baseline and the aspect modified attack vectors stayed fairly static in their level of difficulty and their successful outcome. An attacker would need a significant amount of time and the ability to test multiple candidate strings recovered from the dump against the target RDBMS before finding the payload. It was observed that the use of passwords with lower levels of complexity was more easily captured than if the password had a high level of complexity. Passwords made up of 12 to 16 characters in length utilizing uppercase, lowercase letters, numbers and punctuation characters significantly increased the level of difficulty in capturing the payload.

Capturing the payload was not impossible even with complex passwords, but highly impractical considering most locked down and secure RDBMS's would have login failure monitoring in place to trap multiple failures from a single source. Given time and the proper environment the payload can and would be captured. The significant point here is that this attack vector was unaffected by the use of the aspect methodology proposed in the research.

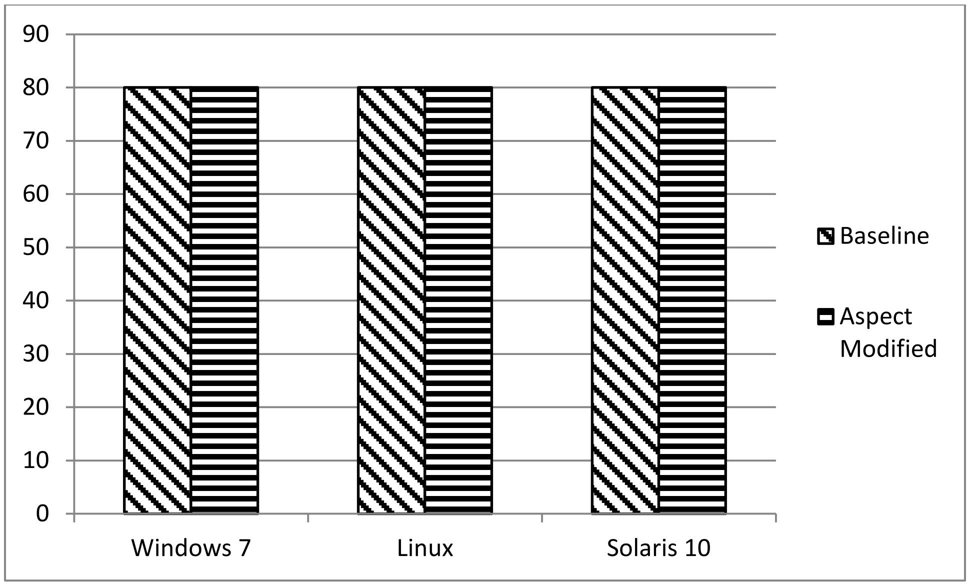


Table 9 Memory Scan Attack Risk Reduction Comparison

Summary of Results

Invoking an Aspect, whose code can be housed outside of the targeted program and called transparently during execution, significantly reduced the risk from attack. Overall, this research was able to effect a 50.82% reduction in risk as measured across all attack vectors. It was noted that changes in risk levels did not vary between platforms, but were surprisingly identical. Basically, the operating made no difference and had no impact on the success or failure of the attack vector. Baseline risk scores of 244 per platform were recorded prior to the introduction of the aspect method and then reduced to risk scores of 120 once an aspect method was introduced, thus supporting and proving that the obfuscation of the application to application connection credentials does in fact significantly reduce risk from attack. The graph in Table 10 demonstrates the levels of risk comparatively.

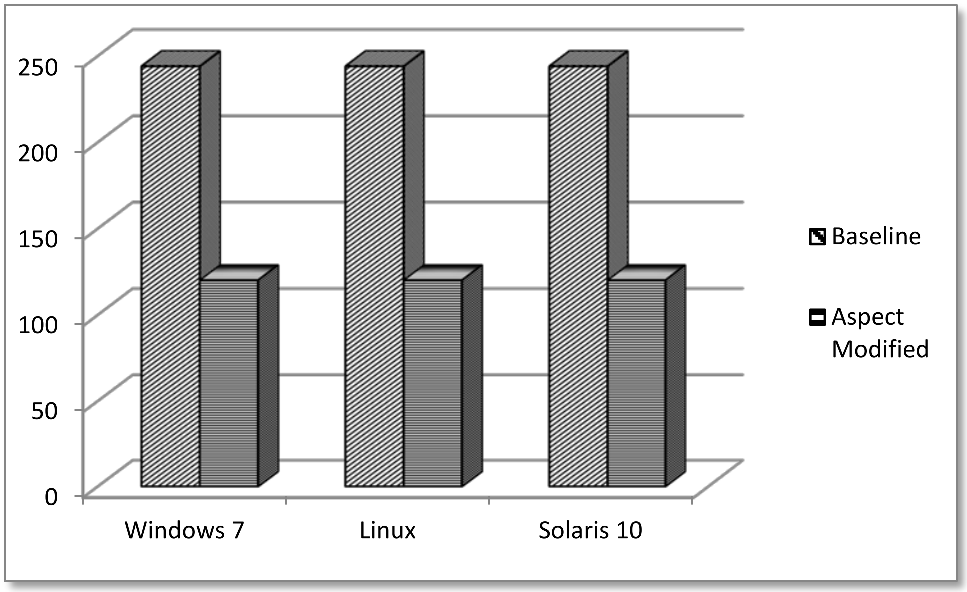


Table 10 Comparative Levels of Risk

It was observed that a higher level of risk reduction was achieved in attack vectors that required lower levels of skill to effect. Conversely, those attack vectors requiring high skill levels saw little or no reduction in risk. A 100% reduction in risk for the probe and logic bomb attack vectors was achieved, while the debugger attack vector achieved a 50% reduction in risk. The memory scan attack vector saw no reduction in risk at all. This is in agreement with the higher skill level required for the attack.

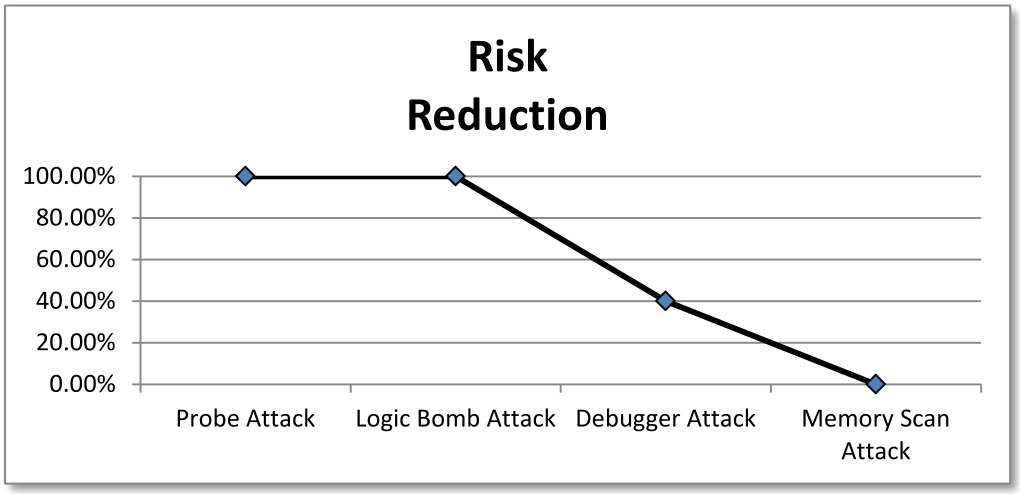


Table 11 Reduction of Risk across Attack Vectors

Introducing an aspect achieved a higher level of obscurity, resulting in a lower level of risk, by implementing concerns that utilized the Composition Filter (CF) model. Obfuscation of the program code and its variables proved to be impossible once stored in system memory. The use of a debugger to examine program code during execution required a higher skill level and the proposed aspect method significantly obstructed the path to the variables, but was not able to block the attack from gaining the payload.

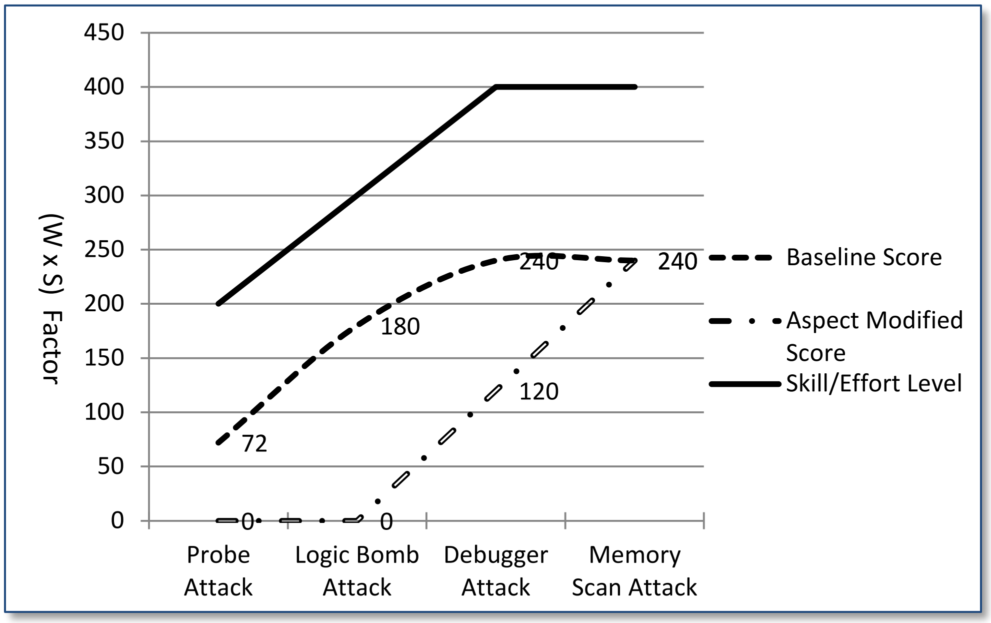


Table 12 Comparative analysis of skill level to attack success

Table 12 demonstrates a comparative analysis of baseline to aspect modified attack vectors as influenced by the skill level of the attack. As the skill level rises, the ability of the aspect method to reduce risk drops.

Considering the two attack methods used in this research, the MATE attack method is by far the more widely used one by attackers (Falcarin et al., 2011), far more so than the memory scan attack method. Further, if one considers only the MATE attack vectors, the Aspect Obfuscation Method reduced risk by 75.61%. Therefore, it can be said without a shadow of doubt that the stated goal of this research has been achieved.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

Analysis of research efforts, auditing and regulatory agency standards and third party products tell us that most people knowledgeable in the field have conceded that the challenge of protecting application-to-application credentials used in interpretive scripts is unsolvable (Chinchani et al., 2005; Chumash & Yao, 2009; Shiflett, 2004; Shmueli et al., 2010; Yu et al., 2009). While the research presented in this work does not claim to eliminate all risk of attack against application-to-application credentials it has proven that the risk can be reduced significantly.

We are told that 17% of all insider attacks are perpetrated by attackers with privileged access (Randazzo et al., 2005). The majority of those attackers do not possess advance programing or forensic analysis skills. They are more crimes of opportunity than not. Yet, very little literature or research has been dedicated to reducing the risk of attacks on application-to-application credentials contained in programs accessible by those attackers. Instead, the focus of auditing and regulatory agencies and third party product researchers has been on mitigating the impact of successful attacks (Franqueira et al., 2010). This author feels, and this dissertation proves out, that the effort would be better spent preventing the attack altogether.

It has long been said that skilled thieves are unstoppable and most locks were designed to keep out honest people and the lessor skilled thieves. The work presented here has proven that it is possible to block attack vectors from attackers with lessor forensic skills and retard attacks from those with medium level skill sets. The development of a framework to obfuscate application-to-application credentials used by interpretive Perl scripts is essential to raising the skill level necessary to successfully carry out an attack. Raising the necessary skill level for successful attack vector directly lowers the risk of exploit.

The key to reducing the risk of any attack is to reduce the attack surface. The research presented here did just that. When credentials are handled by the main body of program code, whether those credentials are hard-coded or returned from a password vault, they are exposed to attack from users with high levels of privilege on many planes. Figure 7 illustrates that by removing the handling of the credentials from the main body of code and relocating them to a secure library (the red circle) the attack surface is significantly reduced.

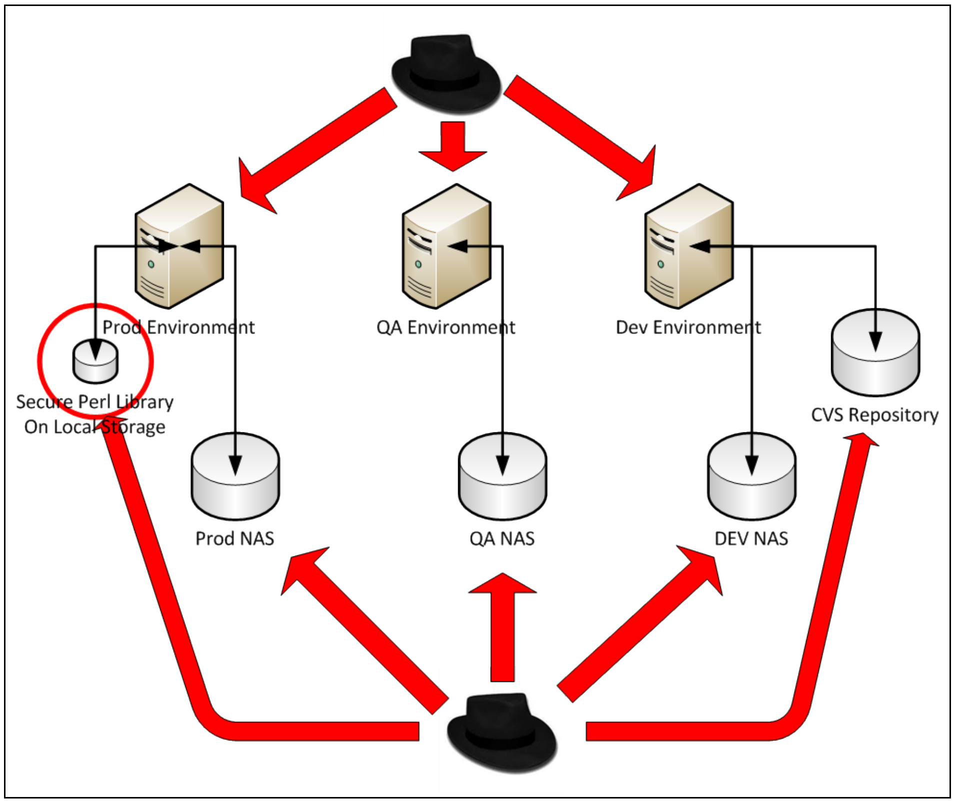


Figure 7 Reducing the attack surface

The framework developed in this work has been able to reduce the risk of MATE attacks by as much as 75.61%. It did, however, fail to reduce the risk from attacks perpetrated by highly skill attackers using a *'low return on investment'* memory scan attack vector. The framework developed in this research was able to effectively reduce *low skill* MATE attack vectors such as probe attacks and logic bomb attack by 100%. The third MATE attack vector explored, the debugger attack vector, had its risk reduced by 50%. This overwhelmingly demonstrates the value of this framework in protecting application-to-application credentials from exploit.

Implications

This dissertation has made a number of contributions to the body of knowledge regarding the protection and handling of application-to-application credentials in interpretive scripts. Most notable is moving the focus of security from post-attack mitigation to pre-attack preventive concepts. Post-attack concepts deal with minimizing the impact and reducing loss from attack, whereas pre-attack concepts focus on attack avoidance and risk reduction.

The framework developed from this research consists of the following recommendations:

- Implement a secure password vault to manage all application-to-application credentials.
- Wrap the password vault product specific code to retrieve those credentials from the vault in a subroutine stored in a secure library or Perl module. The location of which must be secured and monitored real-time.

- Distract the attacker by planting incorrect hard-coded credentials in the script that for all intents and purposes would look like the normal connection credentials needed to establish the connection. These false credentials could also be used as a honey pot to identify attacks in progress and possibly expose the identity of the attacker.
- Invoke an aspect to intercept the subroutine call that establishes the connection to the remote data source from the interpretive script. The aspect will then handle the password vault retrieval logic and replace the supplied credentials with the correct legitimate credentials needed for the connection. It is imperative that the aspect code be housed a library or Perl module, the location of which must be secured. Looking at the code the attacker should not be able to determine that an aspect is being fired off.

Securing application-to-application credentials in interpretive scripts can and will reduce financial impact of successful abuse of privilege attacks. Instead of concentrating on limiting financial loss by focusing on post-attack security; organizations can now focus more on attack avoidance and risk reduction.

Recommendations

This dissertation is the first known to study the impact on risk reduction through the utilization of an aspect to obfuscate application-to-application credentials in a program written in an interpretive scripting language. Aspects have long been used for such useful things as statistics gathering, measuring the number of subroutine calls, analyzing time spent in a subroutine and other crosscutting concerns that span subroutines within a program. A framework addressing the concern of protecting application-to-application credentials, be it crosscutting or not, provides a unique use of aspect technology in protecting those credentials.

Several patent applications mentioned in this research have attempted to move the interception of application-to-application credentials into the network handling logic of the operating system. For obvious reasons already mentioned this approach this is not a good idea. However, additional research could and should be initiated into interfacing the fired-off aspect directly with the operating system to further lock down the credentials in transit from the vault.

Additional studies could and should be initiated that explore the locking down and obfuscating of variables containing credentials in memory. In the last year numerous papers have presented methods and approaches attempting to secure memory through encryption. The work presented by Muller, Freiling, and Dewald (2011) shows promise but does not mesh well with the operating system and causes some issues. Their future work is to focus on developing a special key register resident in the CPU. They further hope to create a third party application that can perform the function on a Windows operating system.

Chhabra, Rogers, Solihin, and Prvulovic (2011) present a hardware and software approach to full system security. Their SecureME solution performs a process that they call memory cloaking. This is quite promising as it allows for the contents of memory locations to be hidden from the OS while the OS is allowed to perform regular memory management functions. The negative impact of their solution is that it could take as much as 13.5% of the system compute resources. If they can refine this process to have less of an impact on the operating system it would be an ideal complement for credential obfuscation via an aspect.

Chhabra and Solihin (2011) present a simple memory encryption solution that is aimed at encrypting memory at the time of system shutdown. This prevents memory scan attacks on non-volatile memory on systems that are at rest and not running. Their approach has tremendous merit and warrants further exploration to see if it is feasible to implement full-time on a running system. Enck et al. (2008) present a similar solution to Chhabra and Solihin, but it also only addresses non-volatile memory. Because memory scan attacks proved successful even with the use of an aspect it is important for future research efforts to focus on and investigate methods to close this gap in security.

Summary

Government regulatory agencies and auditing standards bodies tell us we must never hard-code application-to-application credentials in our scripts. They offer segregation of duties and the use of password vaults as the answer. These agencies also require the implementation of strict guidelines for handling the inevitable data breaches. Researchers conclude there is no way to stop these breaches, universities do not go far enough in considering security concepts in their programming curriculum and programmers more often than not have no concept of what good secure programming methods are.

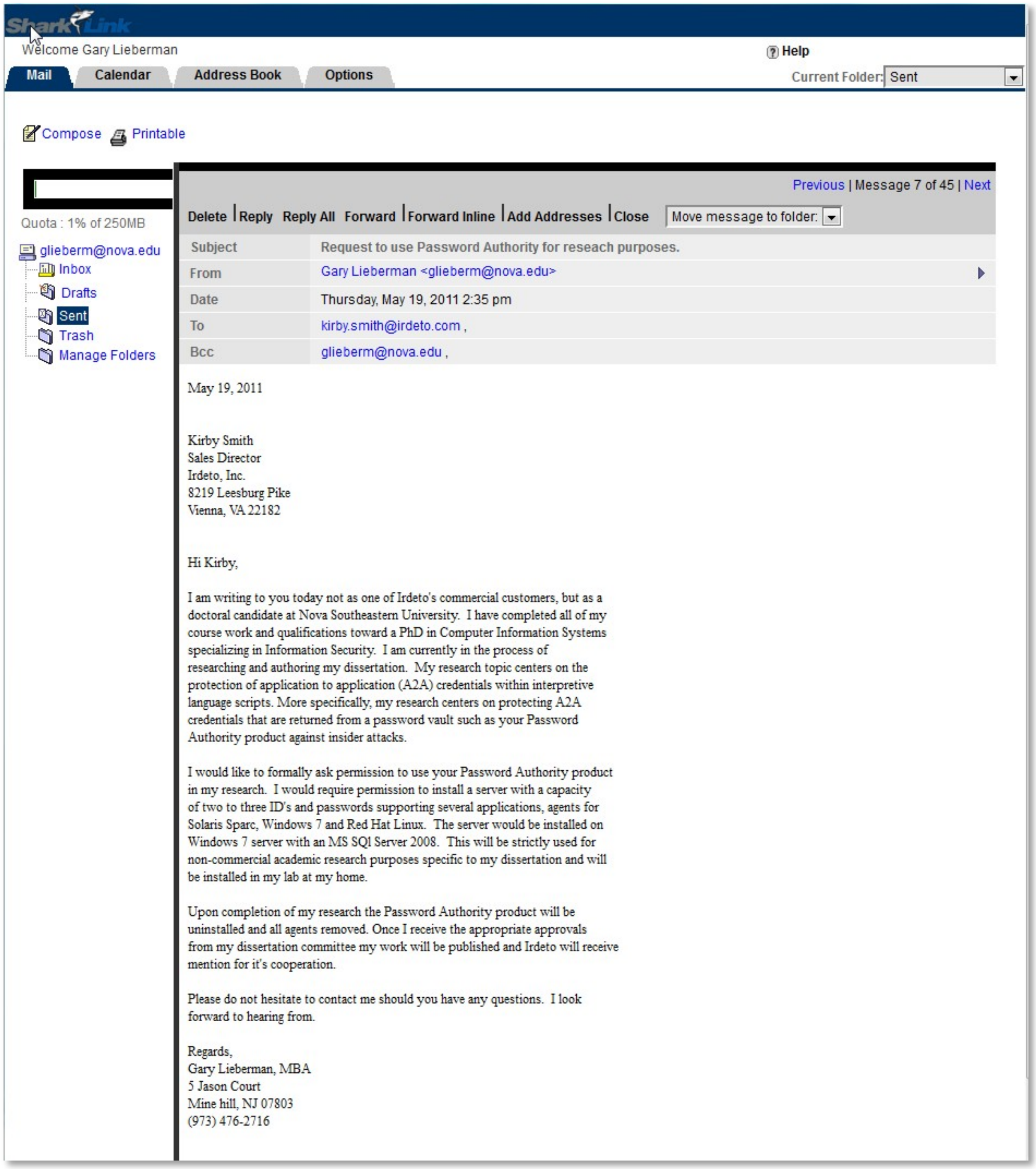
Password vault vendors tout their products as a way to address the insecurity of application-to-application credentials in interpretive scripts. When pressed they reluctantly admit that they have no control on what the script does with the credentials once they are returned to the calling script. It is no wonder that the number of successful insider attacks coming from attackers with privilege access is so high.

The method and framework developed in this dissertation relocates the password retrieval code from the main body of program code, which is under the control of the programmer and accessible by the administrator, to the security of a centralized library module. By obfuscating the handling of the credentials the administrator and the programmer lose their ability to access those credentials. Without ready access to the credentials only the most skilled attackers can capture the attack vector payload.

The goal of this dissertation was to prove that the firing of an aspect and the relocation of password retrieval code from the main body of the program could significantly reduce the risk of attack. This concept was proven as a 50.82% overall reduction of risk was achieved. A 75.61% reduction was achieved for attack vectors dealing directly with program code. Lastly, this dissertation presented the foundation for secure programming methods that can be carried forward and enhanced with future research.

Appendix A

Request for Cloakware License



Subject: Request to use Password Authority for reseach purposes.
From: Gary Lieberman <glieberm@nova.edu>
Date: Thursday, May 19, 2011 2:35 pm
To: kirby.smith@irdeto.com ,
Bcc: glieberm@nova.edu ,

May 19, 2011

Kirby Smith
Sales Director
Irdeto, Inc.
8219 Leesburg Pike
Vienna, VA 22182

Hi Kirby,

I am writing to you today not as one of Irdeto's commercial customers, but as a doctoral candidate at Nova Southeastern University. I have completed all of my course work and qualifications toward a PhD in Computer Information Systems specializing in Information Security. I am currently in the process of researching and authoring my dissertation. My research topic centers on the protection of application to application (A2A) credentials within interpretive language scripts. More specifically, my research centers on protecting A2A credentials that are returned from a password vault such as your Password Authority product against insider attacks.

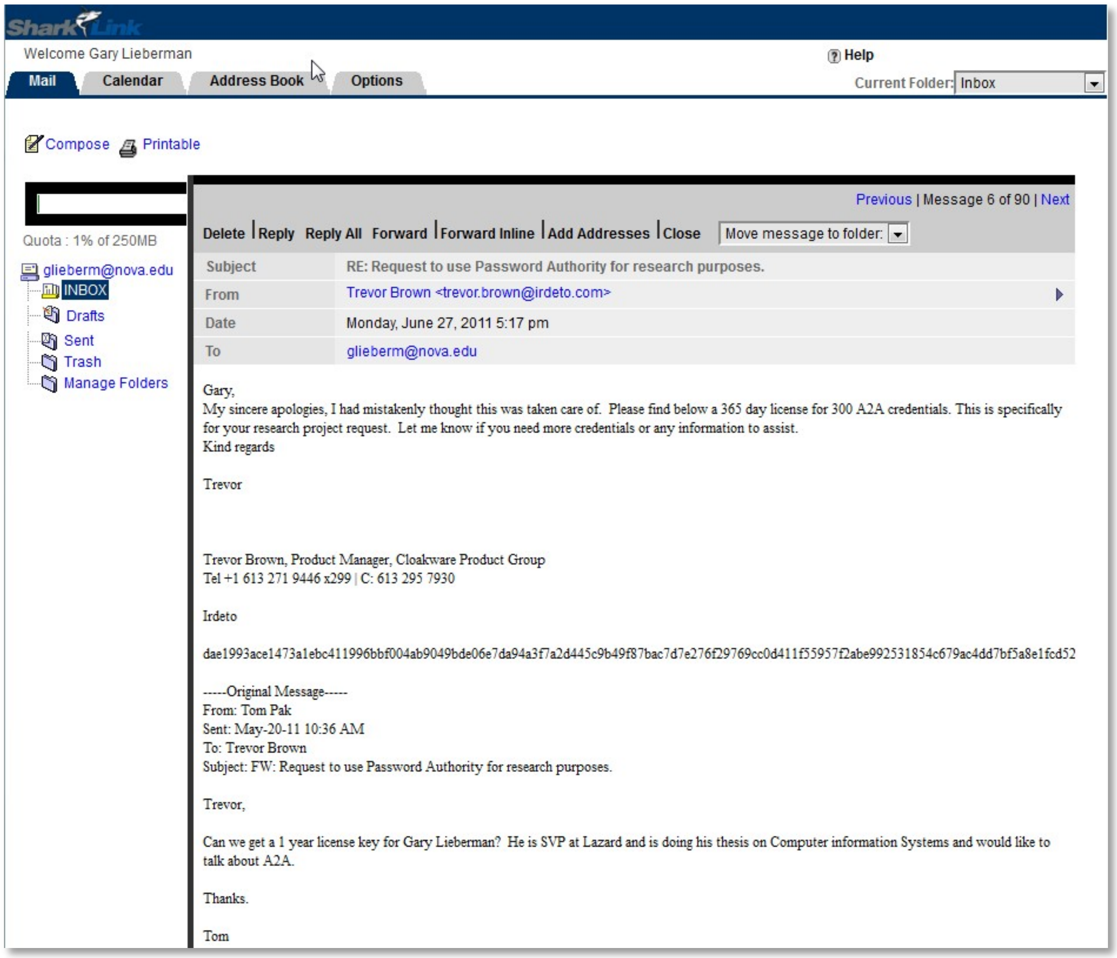
I would like to formally ask permission to use your Password Authority product in my research. I would require permission to install a server with a capacity of two to three ID's and passwords supporting several applications, agents for Solaris Sparc, Windows 7 and Red Hat Linux. The server would be installed on Windows 7 server with an MS SQl Server 2008. This will be strictly used for non-commercial academic research purposes specific to my dissertation and will be installed in my lab at my home.

Upon completion of my research the Password Authority product will be uninstalled and all agents removed. Once I receive the appropriate approvals from my dissertation committee my work will be published and Irdeto will receive mention for it's cooperation.

Please do not hesitate to contact me should you have any questions. I look forward to hearing from.

Regards,
Gary Lieberman, MBA
5 Jason Court
Mine hill, NJ 07803
(973) 476-2716

Figure 8 Cloakware License Request Email



Subject	RE: Request to use Password Authority for research purposes.
From	Trevor Brown <trevor.brown@irdeto.com>
Date	Monday, June 27, 2011 5:17 pm
To	glieberm@nova.edu

Gary,
My sincere apologies, I had mistakenly thought this was taken care of. Please find below a 365 day license for 300 A2A credentials. This is specifically for your research project request. Let me know if you need more credentials or any information to assist.
Kind regards

Trevor

Trevor Brown, Product Manager, Cloakware Product Group
Tel +1 613 271 9446 x299 | C: 613 295 7930

Irdeto

dae1993ace1473a1ebc411996bbf004ab9049bde06e7da94a3f7a2d445c9b49f87bac7d7e276f29769cc0d411f55957f2abe992531854c679ac4dd7bf5a8e1fcd52

-----Original Message-----
From: Tom Pak
Sent: May-20-11 10:36 AM
To: Trevor Brown
Subject: FW: Request to use Password Authority for research purposes.

Trevor,

Can we get a 1 year license key for Gary Lieberman? He is SVP at Lazard and is doing his thesis on Computer information Systems and would like to talk about A2A.

Thanks.

Tom

Figure 9 Cloakware License Use Approval

Appendix B

Hardware Inventory

The following computer equipment will be required to complete this research.

1. Backup Server
    a. HP Compaq 8200 SFF PC
        i. 1 x Quad Core 3.30 GHz Intel i3-2120 CPU
        ii. 12 GB memory
        iii. 500 GB internal hard drive
        iv. Windows 7 Professional 64 bit
        v. Roxio Retrospect V7.7 Backup Software
        vi. HP LT03 Tape Drive
        vii. Quantum DLT Tape Drive
2. Client Servers
    a. Windows 7
        i. HP Elitebook 8440 Notebook PC
            1. 1 x Quad Core Intel 2.76 GHz i7 CPU M620
            2. 4 GB memory
            3. 295 GB internal hard drive
            4. Windows 7 Professional 64 bit
            5. Strawberry Perl version 5.10.1.5
            6. The following base software was installed to facilitate the dumping of live memory:
                a. Microsoft Visual C++ 2010 lC64 Redistributable, version 10.0.30319
                b. Microsoft Visual C++ 2010 x86 Redistributable, version 10.0.30319
                c. Microsoft Windows Performance Toolkit, version 4.8.0
                d. Debugging Tools for Windows (x64), version 612.2.633
                e. Microsoft .NET Framework4 Multi-Targeting Pack, version 4.0.3031g
                f. Application Verifier (lx64), version 4l1078
                g. Microsoft Visual C++ Compilers 2010 Standard - enu - x64, version 10.0.30319

            - h. Microsoft Visual C++ Compilers 2010 Standard - enu - x86, version10.0.30319
            - i. Microsoft Windows SDK for Windows 7 (7.1), version 71.7600.0.30514
            - j. Microsoft Help Viewer 1.0, version 1.0.30319
            - k. Microsoft .NET Framework 4 Extended, version 4.0.30319
            - l. Microsoft Sysinternals Suite, version 02.24.11
            - m. MKStool Kit, version 8
    - b. Solaris 10
        - i. Oracle/Sun SunFire V210
            1. 1 x 1.34 GHz UltraSPARC IIIi CPU
            2. 4 GB memory
            3. 2 x 146gb disk mirrored
            4. Solaris 10
            5. Oracle Database 11g, version 11.2.0.3
            6. Oracle Client 10g, 10.2.0.2
            7. Perl 5.10.0
            8. Cloakware Password Authority version 4.5.0
    - c. openSUSE Linux
        - i. Dell Inspiron 531
            1. 1 x 2.10 GHz AMD Athlon 64 X2 4000 CPU
            2. 4 GB Memory
            3. 250 GB hard drive
            4. openSUSE Linux version 10.3
            5. Perl 5.10.0
3. Storage
    - a. Western Digital ShareSpace 4TB NAS Storage Device
4. Network Equipment
    - a. Sonicwall TZ 100 Network Security Appliance providing DHCP services to the LAN and WLAN.
    - b. 3 x Netgear GS608 8 Port Gigabit Desktop Switches
5. Power Equipment
    - a. 1 x APC SmartUPS 3000
    - b. 1 x APC Back-UPS X5 1500

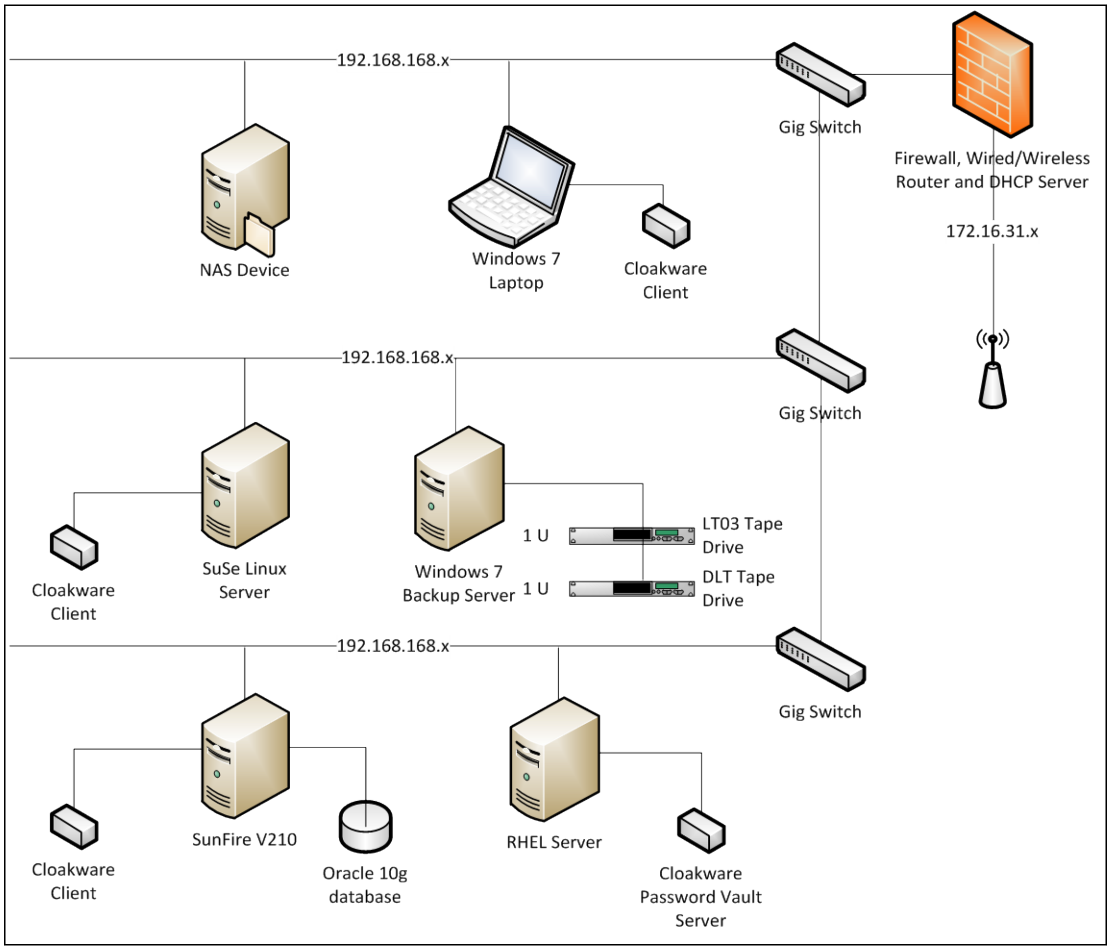


Figure 10 Network Configuration Diagram

Appendix C

Perl Scripts

Baseline Scripts Used for Probe, Debugger and the Memory Scan Attacks

Simple Script Used on Windows 7 and Linux

This is a basic Perl script that has hard-coded application to application credentials. It connects to a MySql database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User,
$Passwd);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
      SELECT      col1, col2
      FROM  novatab
");

for($i=0;$i<100;$i++){
      print "\nLoop $i\n";
      $sth->execute();
      while (my ($col1, $col2) =  $sth->fetchrow_array()){
            print "col1 = $col1, col2 = $col2\n";
      }
      $sth->finish();
      sleep 1;
}
$dbh->disconnect();
exit;
```

Simple Script Used on Solaris 10

This is a basic Perl script that has hard-coded application to application credentials. It connects to an Oracle database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
   SELECT   col1, col2
   FROM     novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
```

Password Vault Script Used on Windows 7 and Linux

This is a essentially the same Perl script described above under “Simple Script Used on Windows 7 and Linux” except the hard-coded application to application credentials have been replaced with a call to a subroutine contained in the NOVA::Passwd Perl module. The new code is highlighted in red below. It connects to a MySql database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","Linux-01")){
	die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary",
$Password);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
	SELECT	col1, col2
	FROM	novatab
");

for($i=0;$i<100;$i++){
	print "\nLoop $i\n";
	$sth->execute();
	while (my ($col1, $col2) =  $sth->fetchrow_array()){
		print "col1 = $col1, col2 = $col2\n";
	}
	$sth->finish();
	sleep 1;
}
$dbh->disconnect();
exit;
```

Password Vault Script Used on Solaris 10

This is a essentially the same Perl script described above (Simple Script Used on Solaris 10) except the hard-coded application to application credentials have been replaced with a call to a subroutine contained in the NOVA::Passwd Perl module. The new code is highlighted in red below. It connects to a Oracle database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","ORPCLK01")){
        die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Password);
die("Cannot open Oracle Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
```

Perl Module Used for Password Vault Access on Windows 7

The following is to code for a Perl module that houses a subroutine (passwdLookup) that performs a call the Cloakware Password Vault. The subroutine expects two arguments to be passed to it. The first is the ID for which the password is being requested. The second is the name of the server being connected to. These two arguments are then combined into a singled string separated by an underscore to form an alias for the ID. The Cloakware Password Vault requires an alias to be passed to it for all password vault requests.

```
package NOVA::Passwd;

our $VERSION = qw ($Revision: 1.0 $)[1];
use vars qw(@ISA @EXPORT);
use Exporter;

use lib "C:/cspm/cloakware/cspmclient/lib";
use CSPM_CLIENT;
use Carp;

@ISA = qw(Exporter);
@EXPORT = qw(passwdLookup);

sub passwdLookup{
        my $cwID = shift;
        my $cwTarget = shift;

        my $errorCode = {
                400 => "Errorcode 400:Success",
                401 => "Errorcode 401: Failed to authenticate with the “.
                       “Password Authority service.",
                402 => "Errorcode 402: Unable to establish connection with “.
                       “client daemon.",
                403 => "Errorcode 403: Not authorized (for client daemon).",
                404 => "Errorcode 404: Unable to establish connection with “.
                       “Password Authority Server.",
                405 => "Errorcode 405: No data found for specified target “.
                        “alias.",
                406 => "Errorcode 406: Application error. See system log for “.
                       “details.",
                407 => "Errorcode 407: Invalid parameters specified.",
                408 => "Errorcode 408: A system error occurred, problem with “.
                       “the “.
                       “client environment. Unable to retrieve environment “.
                        “data.",
                409 => "Errorcode 409: Unauthorized script name.",
                410 => "Errorcode 410: Unauthorized execution path.",
                411 => "Errorcode 411: Unauthorized execution user ID.",
                412 => "Errorcode 412: Unauthorized request server.",
                413 => "Errorcode 413: Client software version is “.
                       “incompatible “.
                       “with the server. ".
                        "This version is no longer supported: upgrade the “.
                        “Password Authority client software.",
                414 => "Errorcode 414: DLL cannot locate exe. (Windows only)",
                415 => "Errorcode 415: DLL internal error occurred. (Windows “.
                       “only)",
                419 => "Errorcode 419: Invalid target alias specified.",
                441 => "Errorcode 441: Invalid file path specified.",
                443 => "Errorcode 443: Client is initializing.",
                445 => "Errorcode 445: Client is updating the encryption key.",
                446 => "Errorcode 446: Authorization mapping validation “.
                       “error. “.
                       “Invalid execution path specified for request script.",
                447 => "Errorcode 447: Authorization mapping validation “
```

```
                        "error. ".
                        "Invalid file path specified for request script.",
                448 => "Errorcode 448: Authorization mapping validation ".
                        "error. ".
                        "Missing request script information.",
                449 => "Errorcode 449: Authorization mapping validation ".
                        "error. ".
                        "Missing hash value for request script."
        };

        if(!defined $cwID or !defined $cwTarget){
                return;
        }else{
                my $cwAlias = $cwID . "_" . $cwTarget;
                my ($cwAnswer, $cwCommand, @cwArray);

                $cwCommand = qq{$GETCR $cwAlias true};
                $cwAnswer = `$cwCommand`;
                @cwArray = split(/\s+/, $cwAnswer);
                if($cwArray[0] ne "400"){
                        carp $errorCode->{$cwArray[0]};
                        return(undef);
                }else{
                        return($cwArray[2]);
                }
        }
}
1:
```

Perl Module Used for Password Vault Access on Linux and Solaris 10

This is essentially the same Perl module described in "Perl Module Used for Password Vault Access on Windows 7" above with the exception of the Cloakware library location and the addition of the environment variables both highlighted in red below.

```
package NOVA::Passwd;

our $VERSION = qw ($Revision: 1.0 $)[1];
use vars qw(@ISA @EXPORT);
use Exporter;

use lib "/opt/cloakware/cspmclient/lib";
use CSPM_CLIENT;
use Carp;
```

```
@ISA = qw(Exporter);
@EXPORT = qw(passwdLookup);

$ENV{'CSPM_CLIENT_HOME'} = "/opt/cloakware";
$ENV{'LD_LIBRARY_PATH'} = "/opt/cloakware/cspmclient/lib:".
                          "/opt/cloakware/cspmclient_thirdparty/java/bin";
$ENV{'CSPM_CLIENT_BIT_TYPE'} = "64";

sub passwdLookup{
        my $cwID = shift;
        my $cwTarget = shift;

        my $errorCode = {
                400 => "Errorcode 400:Success",
                401 => "Errorcode 401: Failed to authenticate with the ".
                       "Password Authority service.",
                402 => "Errorcode 402: Unable to establish connection with ".
                       "client daemon.",
                403 => "Errorcode 403: Not authorized (for client daemon).",
                404 => "Errorcode 404: Unable to establish connection with ".
                       "Password Authority Server.",
                405 => "Errorcode 405: No data found for specified target ".
                        "alias.",
                406 => "Errorcode 406: Application error. See system log for ".
                       "details.",
                407 => "Errorcode 407: Invalid parameters specified.",
                408 => "Errorcode 408: A system error occurred, problem with ".
                       "the ".
                       "client environment. Unable to retrieve environment ".
                        "data.",
                409 => "Errorcode 409: Unauthorized script name.",
                410 => "Errorcode 410: Unauthorized execution path.",
                411 => "Errorcode 411: Unauthorized execution user ID.",
                412 => "Errorcode 412: Unauthorized request server.",
                413 => "Errorcode 413: Client software version is ".
                       "incompatible ".
                       "with the server. ".
                        "This version is no longer supported: upgrade the ".
                        "Password Authority client software.",
                414 => "Errorcode 414: DLL cannot locate exe. (Windows only)",
                415 => "Errorcode 415: DLL internal error occurred. (Windows ".
                       "only)",
                419 => "Errorcode 419: Invalid target alias specified.",
                441 => "Errorcode 441: Invalid file path specified.",
                443 => "Errorcode 443: Client is initializing.",
                445 => "Errorcode 445: Client is updating the encryption key.",
                446 => "Errorcode 446: Authorization mapping validation ".
                       "error. ".
                       "Invalid execution path specified for request script.",
                447 => "Errorcode 447: Authorization mapping validation "
                       "error. ".
                       "Invalid file path specified for request script.",
                448 => "Errorcode 448: Authorization mapping validation ".
                       "error. ".
                       "Missing request script information.",
                449 => "Errorcode 449: Authorization mapping validation ".
                       "error. ".
                       "Missing hash value for request script."
                       "Missing hash value for request script."
        };

        if(!defined $cwID or !defined $cwTarget){
                return;
        }else{
                my $cwAlias = $cwID . "_" . $cwTarget;
                my ($cwAnswer, $cwCommand, @cwArray);

                $cwCommand = qq{$GETCR $cwAlias true};
                $cwAnswer = `$cwCommand`;
                @cwArray = split(/\s+/, $cwAnswer);
                if($cwArray[0] ne "400"){
                        carp $errorCode->{$cwArray[0]};
                        return(undef);
                }else{
                        return($cwArray[2]);
                }
        }
}
1:
```

Baseline Scripts Used for Logic Bomb Attack

Script Used on Windows 7 and Linux

The logic bomb attack consists of the addition of specific code to compromise the credentials contained or handled within the script. The script described under "Simple Script Used on Windows 7 and Linux" was modified to print the credentials to standard out. The code that was added to the script is highlighted below in red.

```
#!/usr/bin/perl

use DBI;
use locale;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User,
$Passwd);
die("Cannot open MySql Connection") if(!$dbh);
#
```

```
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#
my $sth = $dbh->prepare("
      SELECT      col1, col2
      FROM  novatab
");

for($i=0;$i<100;$i++){
      print "\nLoop $i\n";
      $sth->execute();
      while (my ($col1, $col2) =  $sth->fetchrow_array()){
            print "col1 = $col1, col2 = $col2\n";
      }
      $sth->finish();
      sleep 1;
}
$dbh->disconnect();
exit;
```

Script Used on Solaris 10

The logic bomb attack consists of the addition of specific code to compromise the credentials contained or handled within the script. The script described under “Simple Script Used on Solaris 10” was modified to print the credentials to standard out. The code that was added to the script is highlighted below in red.

```
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
#
# Added logic bomb to print the password
#
print "$Passwd\n";
```

```
#
# End logic bomb code
#
my $sth = $dbh->prepare("
      SELECT      col1, col2
     FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
```

Aspect Modified Scripts Used for Probe, Debugger and the Memory Scan Attacks

Simple Password Vault Script Used on Windows 7 and Linux

This basic Perl script that has hard-coded application to application credentials was left as is except for the pragma call added to invoke the aspect shown in red below.   The call to DBI->connect is intercepted and the password parameter replaced with a password retrieved from the vault. It connects to a MySql database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$User = 'gary';
$Passwd = 'novaphd';
```

```
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User,
$Passwd);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
      SELECT      col1, col2
      FROM  novatab
");

for($i=0;$i<100;$i++){
      print "\nLoop $i\n";
      $sth->execute();
      while (my ($col1, $col2) =  $sth->fetchrow_array()){
            print "col1 = $col1, col2 = $col2\n";
      }
      $sth->finish();
      sleep 1;
}
$dbh->disconnect();
exit;
```

Simple Password vault Script Used on Solaris 10

This is basic Perl script that has hard-coded application to application credentials left as is except for the pragma call added to invoke the aspect shown in red below. It connects to a Oracle database server and performs a simple query in a loop of 100 iterations.

```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;


$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
      SELECT      col1, col2
     FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
```

Aspect Modified Perl Module Used for Password Vault Access on Windows 7

The following is to code for a Perl module that houses two subroutines that perform calls the Cloakware Password Vault. The subroutine lookupPaswd() expects two arguments to be passed to it. The first is the ID for which the password is being requested. The second is the name of the server being connected to. These two arguments are then combined into a singled string separated by an underscore to form an alias for the ID. The Cloakware Password Vault requires an alias to be passed to it for all password vault requests.

The second subroutine, passwdAspect() invokes an aspect to intercept calls to the DBI->connect subroutine and replaces the supplied password with one retrieved from the password vault. The aspect is invoke automatically when the Perl module is loaded at execution time.

```
package NOVA::Passwd;

our $VERSION = qw ($Revision: 1.0 $)[1];
use vars qw(@ISA @EXPORT);
use Exporter;

use lib "C:/cspm/cloakware/cspmclient/lib";
use CSPM_CLIENT;
use Carp;

@ISA = qw(Exporter);
@EXPORT = qw(passwdLookup passwdAspect);

sub passwdLookup{
        my $cwID = shift;
        my $cwTarget = shift;

        my $errorCode = {
                400 => "Errorcode 400:Success",
                401 => "Errorcode 401: Failed to authenticate with the ".
                       "Password Authority service.",
                402 => "Errorcode 402: Unable to establish connection with ".
                       "client daemon.",
                403 => "Errorcode 403: Not authorized (for client daemon).",
                404 => "Errorcode 404: Unable to establish connection with ".
                       "Password Authority Server.",
                405 => "Errorcode 405: No data found for specified target ".
                        "alias.",
                406 => "Errorcode 406: Application error. See system log for ".
                       "details.",
                407 => "Errorcode 407: Invalid parameters specified.",
                408 => "Errorcode 408: A system error occurred, problem with ".
                       "the ".
                       "client environment. Unable to retrieve environment ".
                        "data.",
                409 => "Errorcode 409: Unauthorized script name.",
                410 => "Errorcode 410: Unauthorized execution path.",
                411 => "Errorcode 411: Unauthorized execution user ID.",
                412 => "Errorcode 412: Unauthorized request server.",
                413 => "Errorcode 413: Client software version is ".
                       "incompatible ".
                       "with the server. ".
                        "This version is no longer supported: upgrade the ".
                        "Password Authority client software.",
                414 => "Errorcode 414: DLL cannot locate exe. (Windows only)",
                415 => "Errorcode 415: DLL internal error occurred. (Windows ".
                       "only)",
                419 => "Errorcode 419: Invalid target alias specified.",
                441 => "Errorcode 441: Invalid file path specified.",
                443 => "Errorcode 443: Client is initializing.",
                445 => "Errorcode 445: Client is updating the encryption key.",
                446 => "Errorcode 446: Authorization mapping validation ".
                       "error. ".
                       "Invalid execution path specified for request script.",
                447 => "Errorcode 447: Authorization mapping validation "
                       "error. ".
                       "Invalid file path specified for request script.",
                448 => "Errorcode 448: Authorization mapping validation ".
                       "error. ".
```

```
                        "Missing request script information.",
                449 => "Errorcode 449: Authorization mapping validation ".
                        "error. ".
                        "Missing hash value for request script."
        };

        if(!defined $cwID or !defined $cwTarget){
                return;
        }else{
                my $cwAlias = $cwID . "_" . $cwTarget;
                my ($cwAnswer, $cwCommand, @cwArray);

                $cwCommand = qq{$GETCR $cwAlias true};
                $cwAnswer = `$cwCommand`;
                @cwArray = split(/\s+/, $cwAnswer);
                if($cwArray[0] ne "400"){
                        carp $errorCode->{$cwArray[0]};
                        return(undef);
                }else{
                        return($cwArray[2]);
                }
        }
}

sub passwdAspect{
        before {
                my $context = shift;
                my ($cwTagert, $cwID, $cwPasswd);

                my @params = $context->params;
                $cwTarget = $params[1];
                $cwTarget =~ s/^.*=//;
                $cwID = $params[2];
                unless($params[3] = passwdLookup($cwID,$cwTarget)){
                        print STDERR "\nThe passwd $cwID not found in
Cloakware,".
                        "switching to passthrough mode\n";
                }else{
                        $context->params($params[0],$params[1],$params[2],
                                    $params[3]);
                }
        } call ('DBI::connect');
}

passwdAspect;
1:
```

Aspect Modified Perl Module Used for Password Vault Access on Linux and Solaris 10

This is essentially the same Perl module described in “Aspect Modified Perl Module Used for Password Vault Access on Windows 7” above with the exception of the Cloakware library location and the addition of the environment variables both highlighted in red below.

```
package NOVA::Passwd;

our $VERSION = qw ($Revision: 1.0 $)[1];
use vars qw(@ISA @EXPORT);
use Exporter;

use lib "/opt/cloakware/cspmclient/lib";
use CSPM_CLIENT;
use Carp;

@ISA = qw(Exporter);
@EXPORT = qw(passwdLookup passwdAspect);

$ENV{'CSPM_CLIENT_HOME'} = "/opt/cloakware";
$ENV{'LD_LIBRARY_PATH'} = "/opt/cloakware/cspmclient/lib:”.
                          “/opt/cloakware/cspmclient_thirdparty/java/bin";
$ENV{'CSPM_CLIENT_BIT_TYPE'} = "64";

sub passwdLookup{
        my $cwID = shift;
        my $cwTarget = shift;

        my $errorCode = {
                400 => "Errorcode 400:Success",
                401 => "Errorcode 401: Failed to authenticate with the “.
                       “Password Authority service.",
                402 => "Errorcode 402: Unable to establish connection with “.
                       “client daemon.",
                403 => "Errorcode 403: Not authorized (for client daemon).",
                404 => "Errorcode 404: Unable to establish connection with “.
                       “Password Authority Server.",
                405 => "Errorcode 405: No data found for specified target “.
                        “alias.",
                406 => "Errorcode 406: Application error. See system log for “.
                       “details.",
                407 => "Errorcode 407: Invalid parameters specified.",
                408 => "Errorcode 408: A system error occurred, problem with “.
                       “the “.
                       “client environment. Unable to retrieve environment “.
                        “data.",
                409 => "Errorcode 409: Unauthorized script name.",
                410 => "Errorcode 410: Unauthorized execution path.",
                411 => "Errorcode 411: Unauthorized execution user ID.",
                412 => "Errorcode 412: Unauthorized request server.",
                413 => "Errorcode 413: Client software version is ".
                       "incompatible ".
                       "with the server. ".
                        "This version is no longer supported: upgrade the ".
                        "Password Authority client software.",
                414 => "Errorcode 414: DLL cannot locate exe. (Windows only)",
                415 => "Errorcode 415: DLL internal error occurred. (Windows ".
                       "only)",
                419 => "Errorcode 419: Invalid target alias specified.",
                441 => "Errorcode 441: Invalid file path specified.",
                443 => "Errorcode 443: Client is initializing.",
                445 => "Errorcode 445: Client is updating the encryption key.",
                446 => "Errorcode 446: Authorization mapping validation ".
                       "error. ".
                       "Invalid execution path specified for request script.",
                447 => "Errorcode 447: Authorization mapping validation "
                       "error. ".
                       "Invalid file path specified for request script.",
                448 => "Errorcode 448: Authorization mapping validation ".
                       "error. ".
                       "Missing request script information.",
                449 => "Errorcode 449: Authorization mapping validation ".
                       "error. ".
        };

        if(!defined $cwID or !defined $cwTarget){
                return;
        }else{
                my $cwAlias = $cwID . "_" . $cwTarget;
                my ($cwAnswer, $cwCommand, @cwArray);

                $cwCommand = qq{$GETCR $cwAlias true};
                $cwAnswer = `$cwCommand`;
                @cwArray = split(/\s+/, $cwAnswer);
                if($cwArray[0] ne "400"){
                        carp $errorCode->{$cwArray[0]};
                        return(undef);
                }else{
                        return($cwArray[2]);
                }
        }
}

sub passwdAspect{
        before {
                my $context = shift;
                my ($cwTagert, $cwID, $cwPasswd);

                my @params = $context->params;
                $cwTarget = $params[1];
                $cwTarget =~ s/^.*=//;
                $cwID = $params[2];
```

```
               unless($params[3] = passwdLookup($cwID,$cwTarget)){
                      print STDERR "\nThe passwd $cwID not found in
Cloakware,”.
                      “switching to passthrough mode\n";
               }else{
                      $context->params($params[0],$params[1],$params[2],
                               $params[3]);
               }
       } call ('DBI::connect');
}

passwdAspect;
1:
```

Aspect Modified Scripts Used for Logic Bomb Attack

Script Used on Windows 7 and Linux

The logic bomb attack consists of the addition of specific code to compromise the credentials contained or handled within the script. The script described under “Simple Script Used on Windows 7 and Linux” was modified to print the credentials to standard out. The code that was added to the script is highlighted below in red.

```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;


$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User,
$Passwd);
die("Cannot open MySql Connection") if(!$dbh);
#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
```

```
#
my $sth = $dbh->prepare("
      SELECT      col1, col2
      FROM  novatab
");

for($i=0;$i<100;$i++){
      print "\nLoop $i\n";
      $sth->execute();
      while (my ($col1, $col2) =  $sth->fetchrow_array()){
            print "col1 = $col1, col2 = $col2\n";
      }
      $sth->finish();
      sleep 1;
}
$dbh->disconnect();
exit;
```

Script Used on Solaris 10

The logic bomb attack consists of the addition of specific code to compromise the credentials contained or handled within the script. The script described under "Simple Script Used on Solaris 10" was modified to print the credentials to standard out. The code that was added to the script is highlighted below in red.

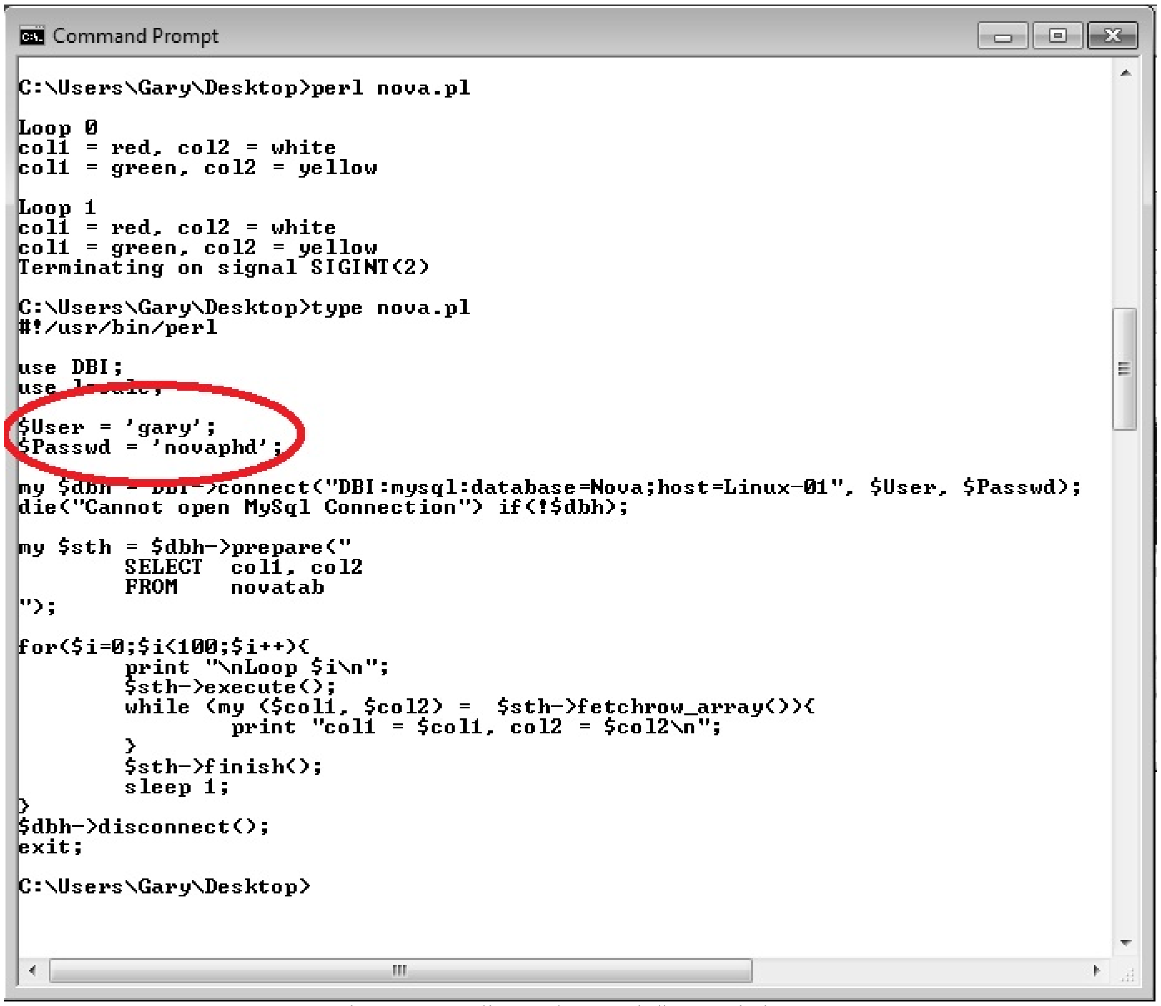
```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;


$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
```

```
#
my $sth = $dbh->prepare("
      SELECT      col1, col2
     FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
```

Appendix D

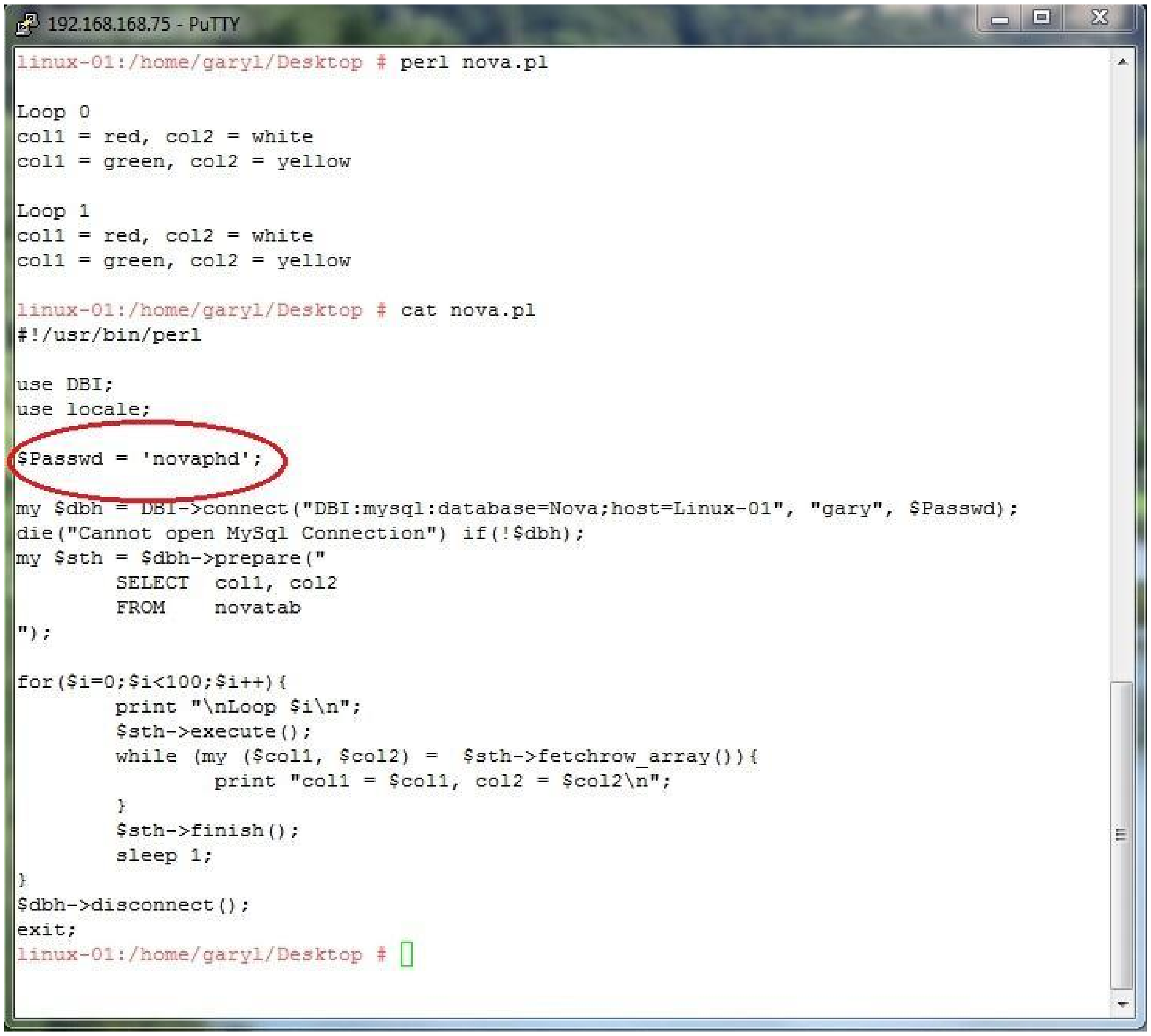
Attack Vector Screenshots

Baseline Probe Attack Vector Screenshots



```
C:\Users\Gary\Desktop>perl nova.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow
Terminating on signal SIGINT(2)

C:\Users\Gary\Desktop>type nova.pl
#!/usr/bin/perl

use DBI;
use locale;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User, $Passwd);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

C:\Users\Gary\Desktop>
```

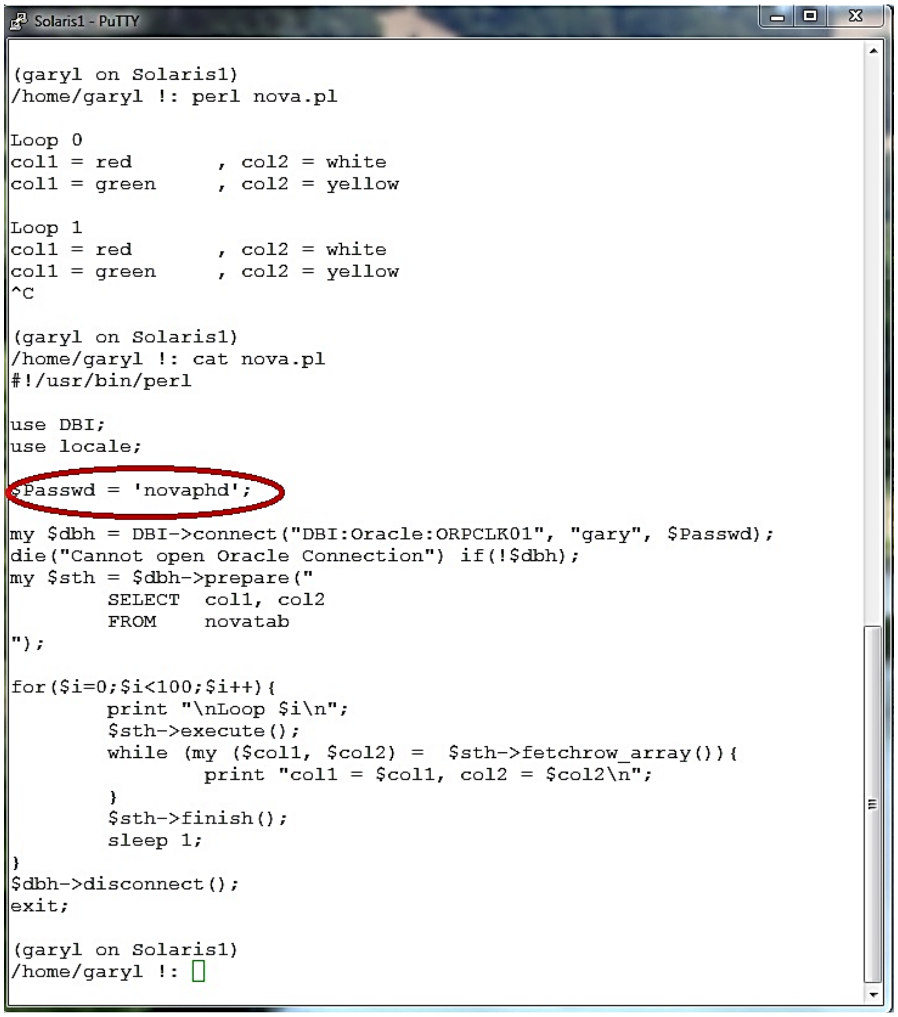
Figure 11 Baseline Probe Attack #1 on Windows 7



192.168.168.75 - PuTTY

```
linux-01:/home/garyl/Desktop # perl nova.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

linux-01:/home/garyl/Desktop # cat nova.pl
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Passwd);
die("Cannot open MySql Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
linux-01:/home/garyl/Desktop #
```

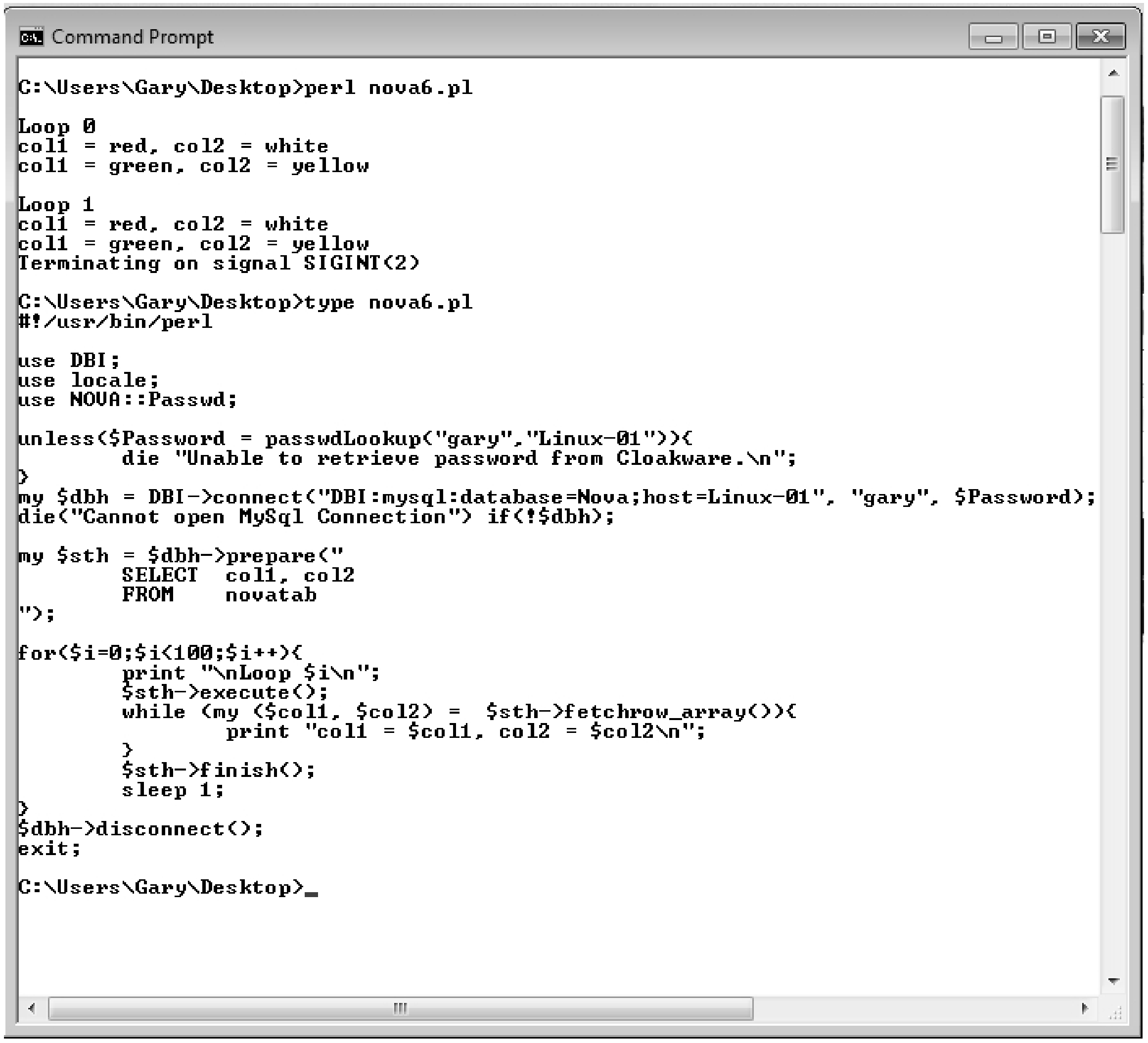
Figure 12 Baseline Probe Attack #1 on Linux



```
(garyl on Solaris1)
/home/garyl !: perl nova.pl

Loop 0
col1 = red       , col2 = white
col1 = green     , col2 = yellow

Loop 1
col1 = red       , col2 = white
col1 = green     , col2 = yellow
^C

(garyl on Solaris1)
/home/garyl !: cat nova.pl
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

(garyl on Solaris1)
/home/garyl !:
```

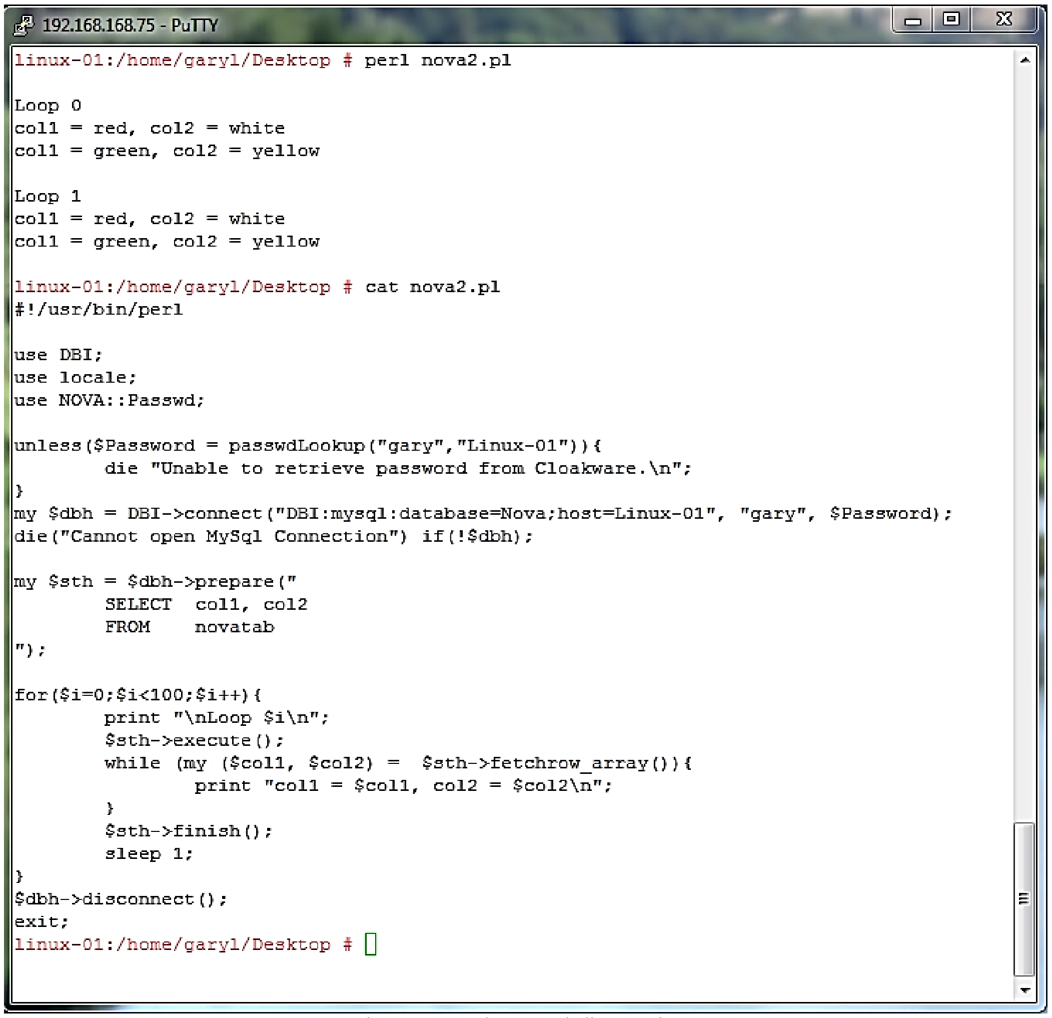
Figure 13 Baseline Probe Attack #1 on Solaris 10

The screenshot showed in figures 11, 12 and 13 show the test script running on all three platforms and being terminated after two iterations. Then the probe attack was carried out against the script. The payload of the probe attack vector is shown circled in red.


Command Prompt

```
C:\Users\Gary\Desktop>perl nova6.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow
Terminating on signal SIGINT(2)

C:\Users\Gary\Desktop>type nova6.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","Linux-01")){
        die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Password);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

C:\Users\Gary\Desktop>_
```

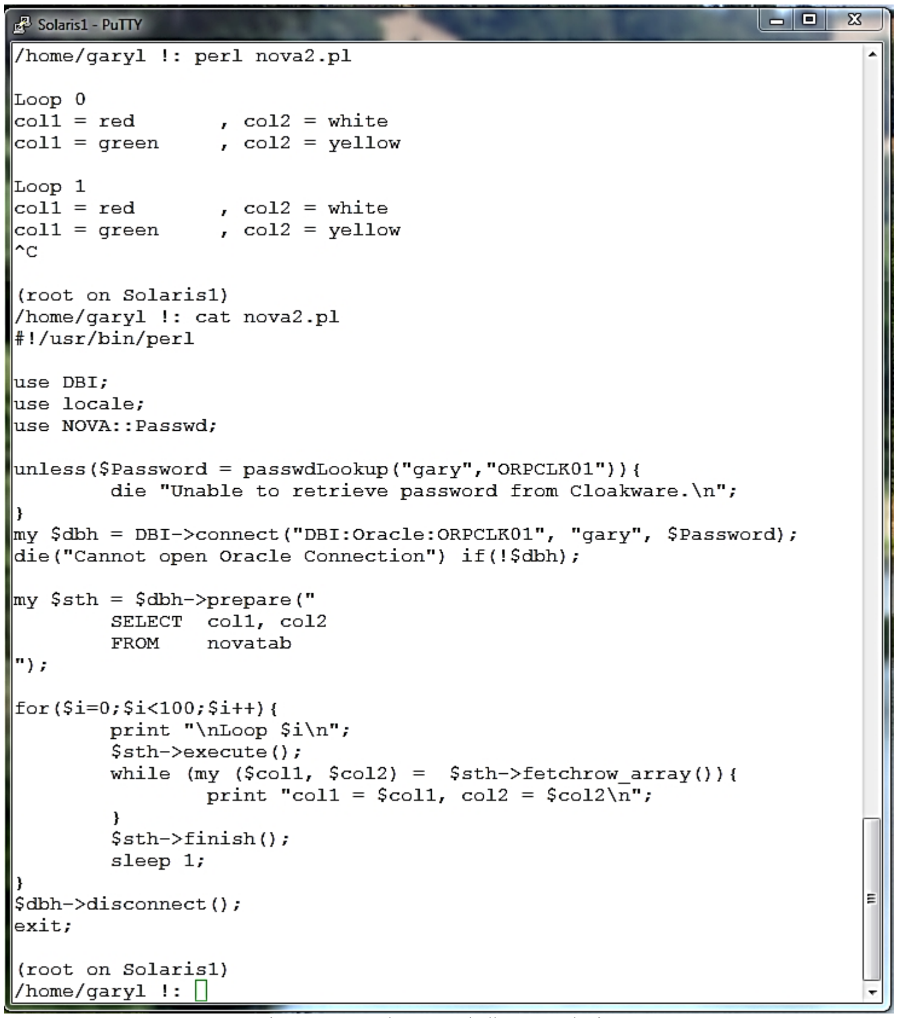
Figure 14 Probe Attack #2 on Windows 7



192.168.168.75 - PuTTY

```
linux-01:/home/garyl/Desktop # perl nova2.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

linux-01:/home/garyl/Desktop # cat nova2.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","Linux-01")){
        die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Password);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
linux-01:/home/garyl/Desktop #
```

Figure 15 Probe Attack #2 on Linux

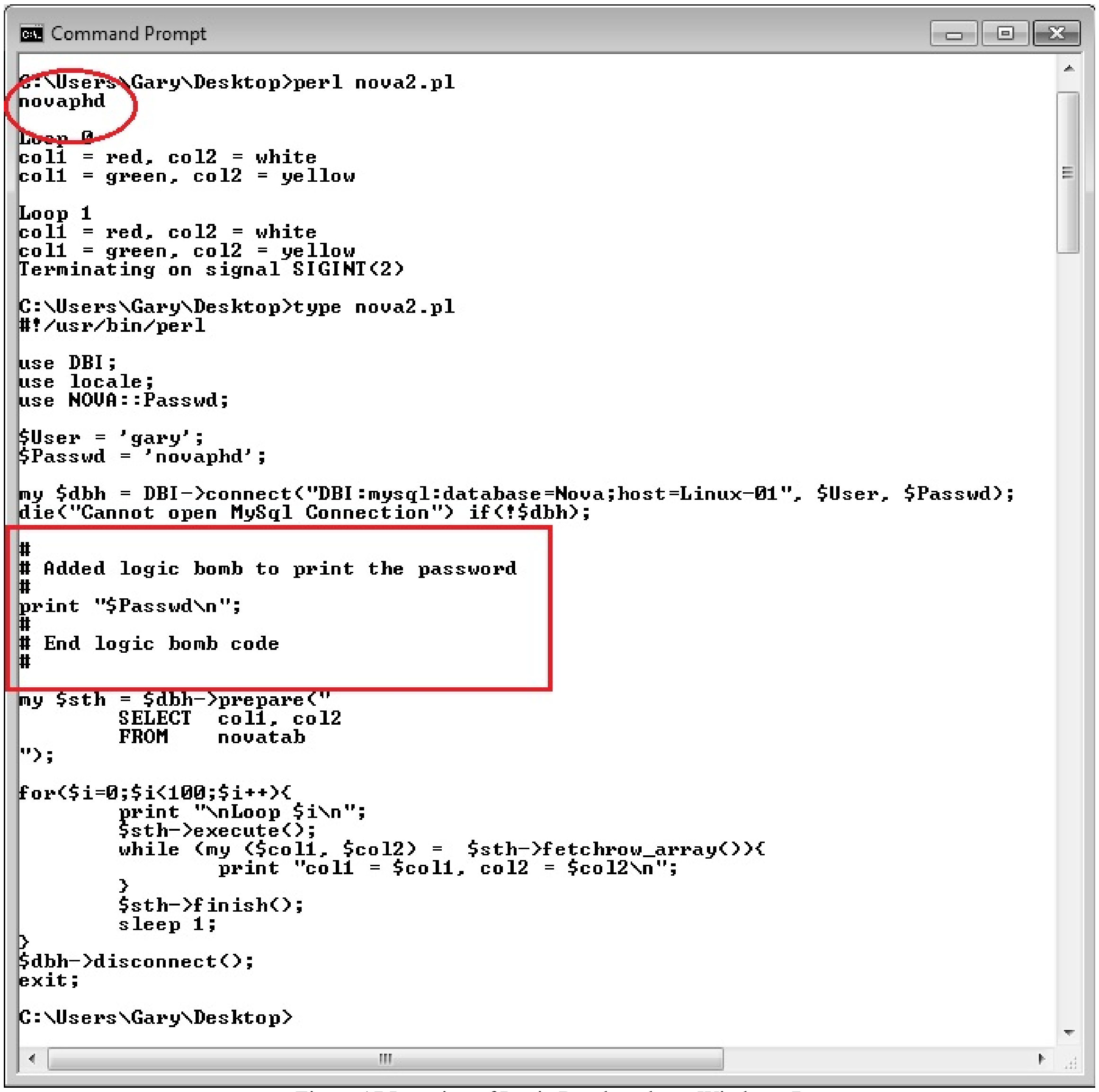

Solaris1 - PuTTY

```
/home/garyl !: perl nova2.pl

Loop 0
col1 = red       , col2 = white
col1 = green     , col2 = yellow

Loop 1
col1 = red       , col2 = white
col1 = green     , col2 = yellow
^C

(root on Solaris1)
/home/garyl !: cat nova2.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","ORPCLK01")){
        die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Password);
die("Cannot open Oracle Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

(root on Solaris1)
/home/garyl !: 
```

Figure 16 Probe Attack #2 on Solaris 10

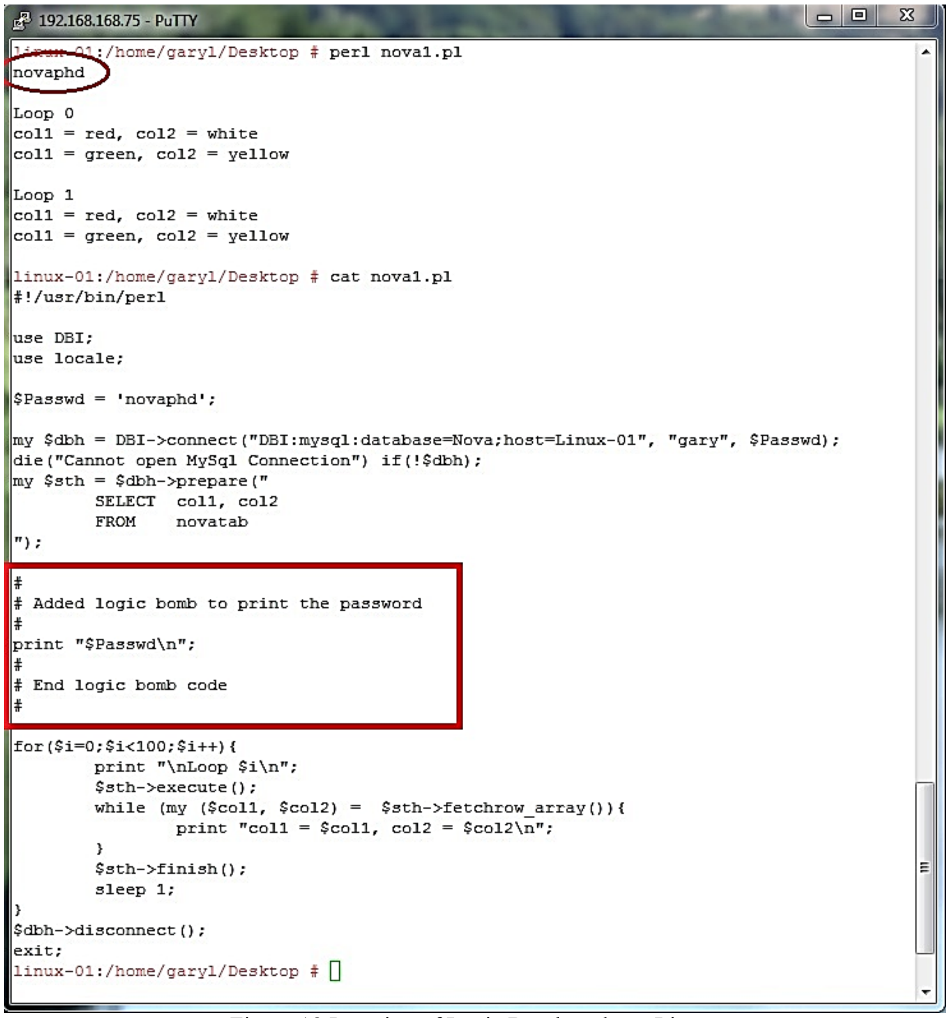
Figures 14, 15 and 16 show an attempted probe attack on the script utilizing the passwdLookup subroutine held in the NOVA::Passwd Perl module. The probe attack vector was carried out on all three platforms and was not successful in capturing the payload.

Baseline Logic Bomb Attack Vector Screenshots


Command Prompt

```
C:\Users\Gary\Desktop>perl nova2.pl
novaphd

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow
Terminating on signal SIGINT(2)

C:\Users\Gary\Desktop>type nova2.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User, $Passwd);
die("Cannot open MySql Connection") if(!$dbh);

#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

C:\Users\Gary\Desktop>
```

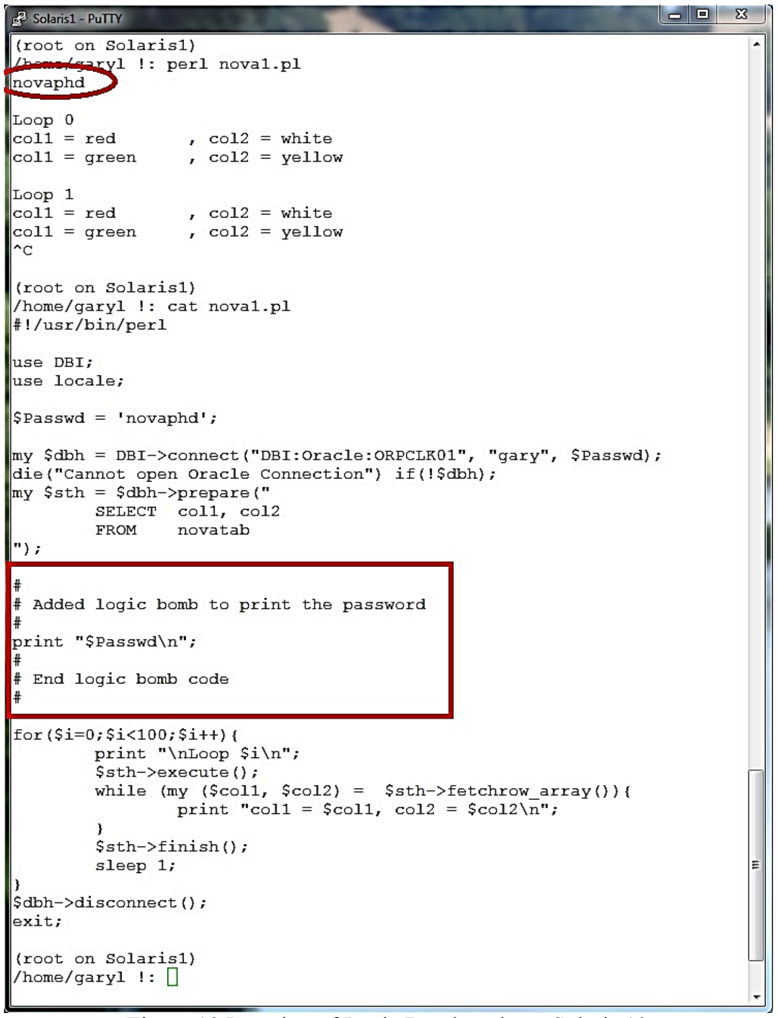
Figure 17 Insertion of Logic Bomb code on Windows 7



192.168.168.75 - PuTTY

```
linux-01:/home/garyl/Desktop # perl nova1.pl
novaphd

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

linux-01:/home/garyl/Desktop # cat nova1.pl
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Passwd);
die("Cannot open MySql Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
linux-01:/home/garyl/Desktop #
```

Figure 18 Insertion of Logic Bomb code on Linux.

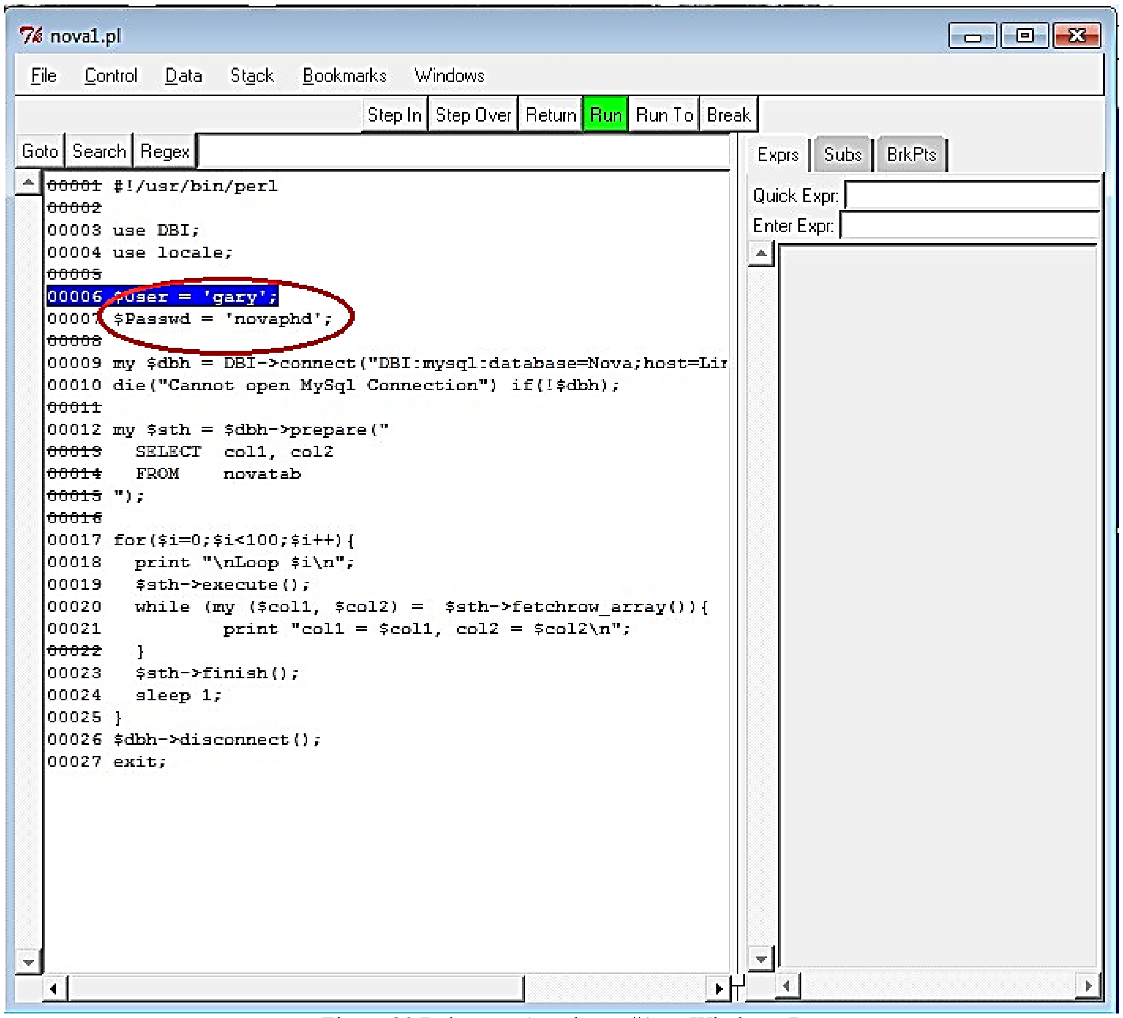


```
(root on Solaris1)
/home/garyl !: perl nova1.pl
novaphd

Loop 0
col1 = red       , col2 = white
col1 = green     , col2 = yellow

Loop 1
col1 = red       , col2 = white
col1 = green     , col2 = yellow
^C

(root on Solaris1)
/home/garyl !: cat nova1.pl
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

(root on Solaris1)
/home/garyl !:
```

Figure 19 Insertion of Logic Bomb code on Solaris 10

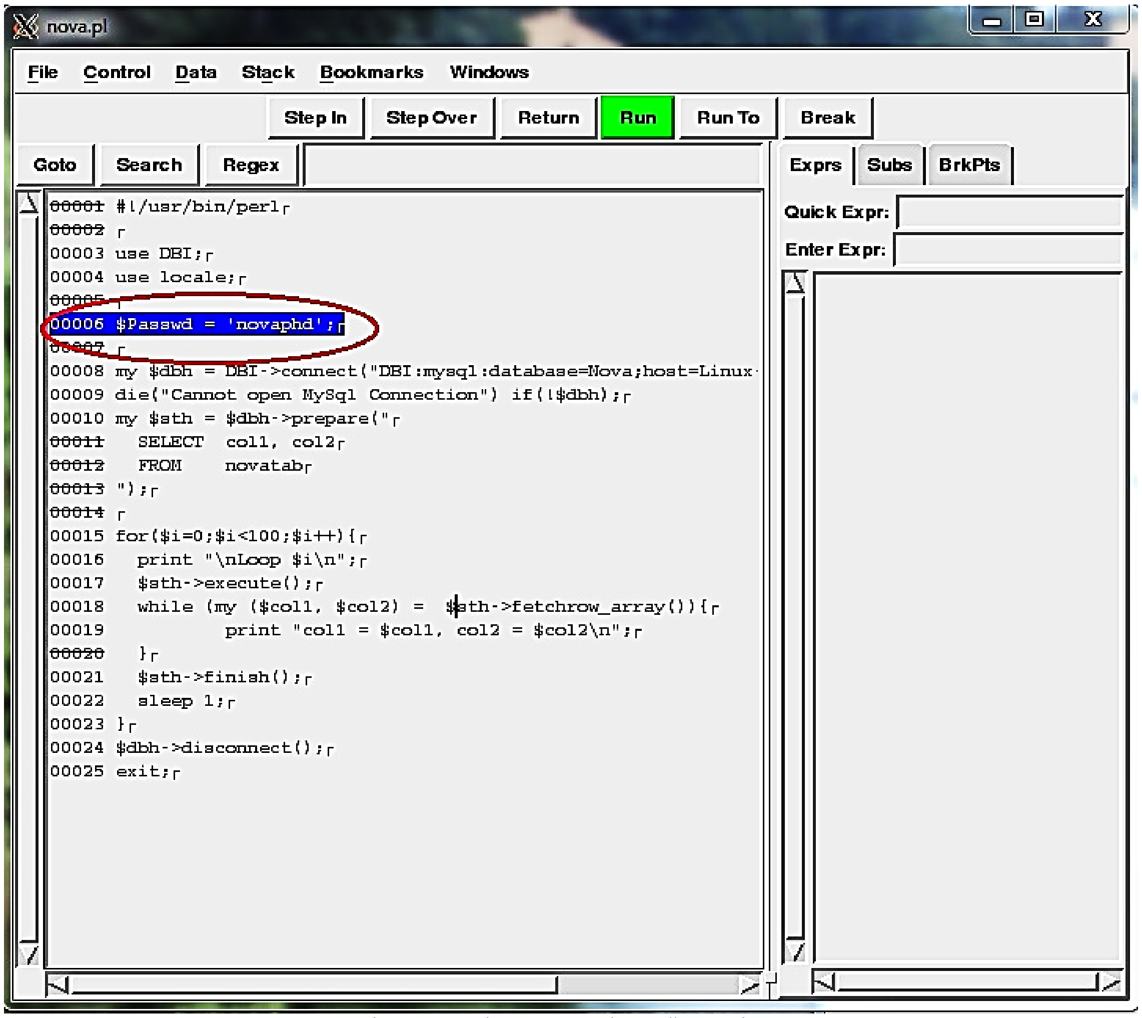
Figures 17, 18 and 19 show a successful logic bomb attack vector carried out on all three platforms. The code encased in the red rectangle is the logic bomb code place in the script. The payload form the logic bomb attack vector is circled in red.

Baseline Debugger Attack Vector Screenshots


novа1.pl
File Control Data Stack Bookmarks Windows
Step In Step Over Return Run Run To Break
Goto Search Regex
Exprs Subs BrkPts
Quick Expr:
Enter Expr:

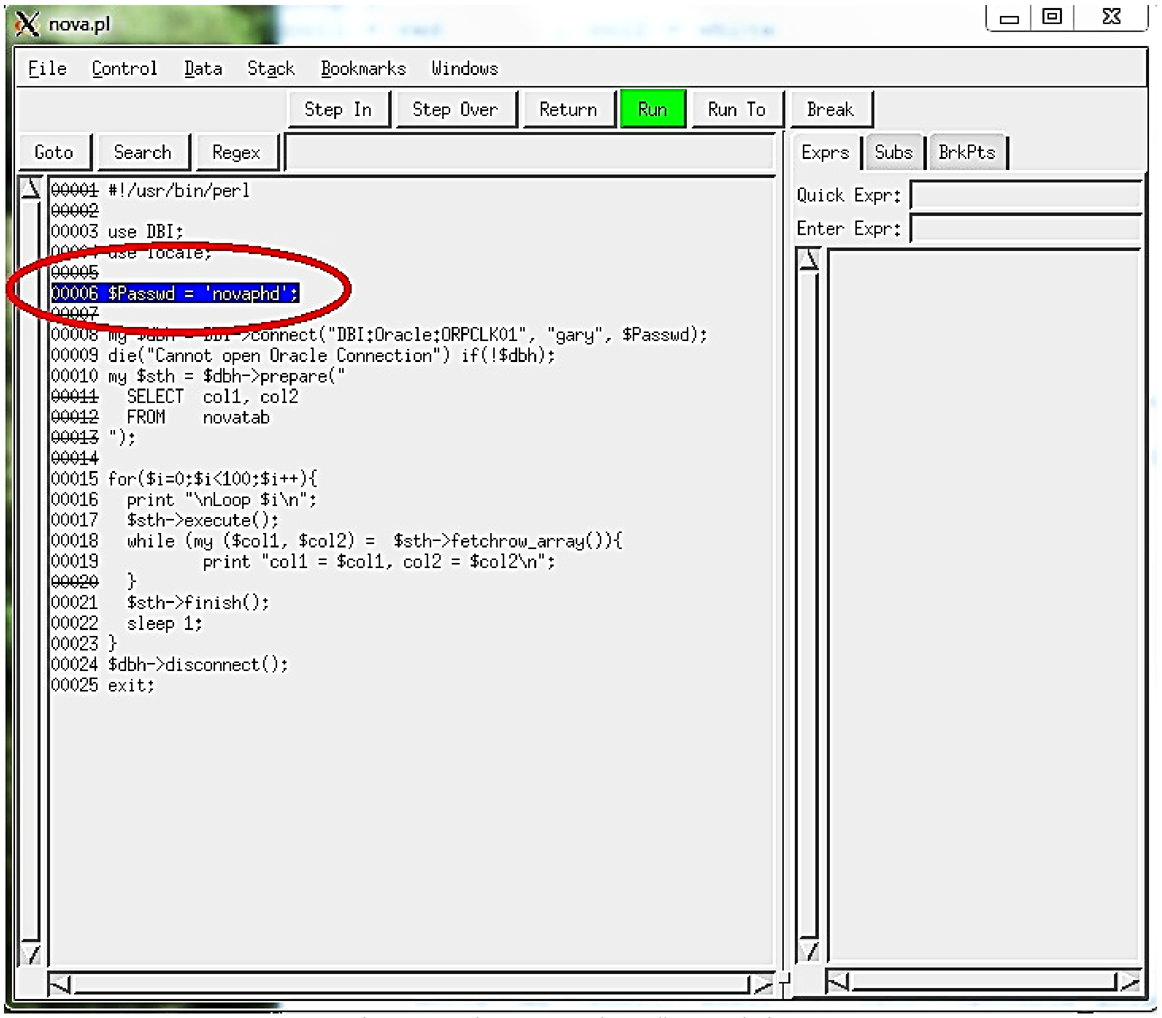
```
#!/usr/bin/perl

use DBI;
use locale;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Lir
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

Figure 20 Debugger Attack run #1 on Windows 7



```
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux
die("Cannot open MySql Connection") if(!$dbh);
my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

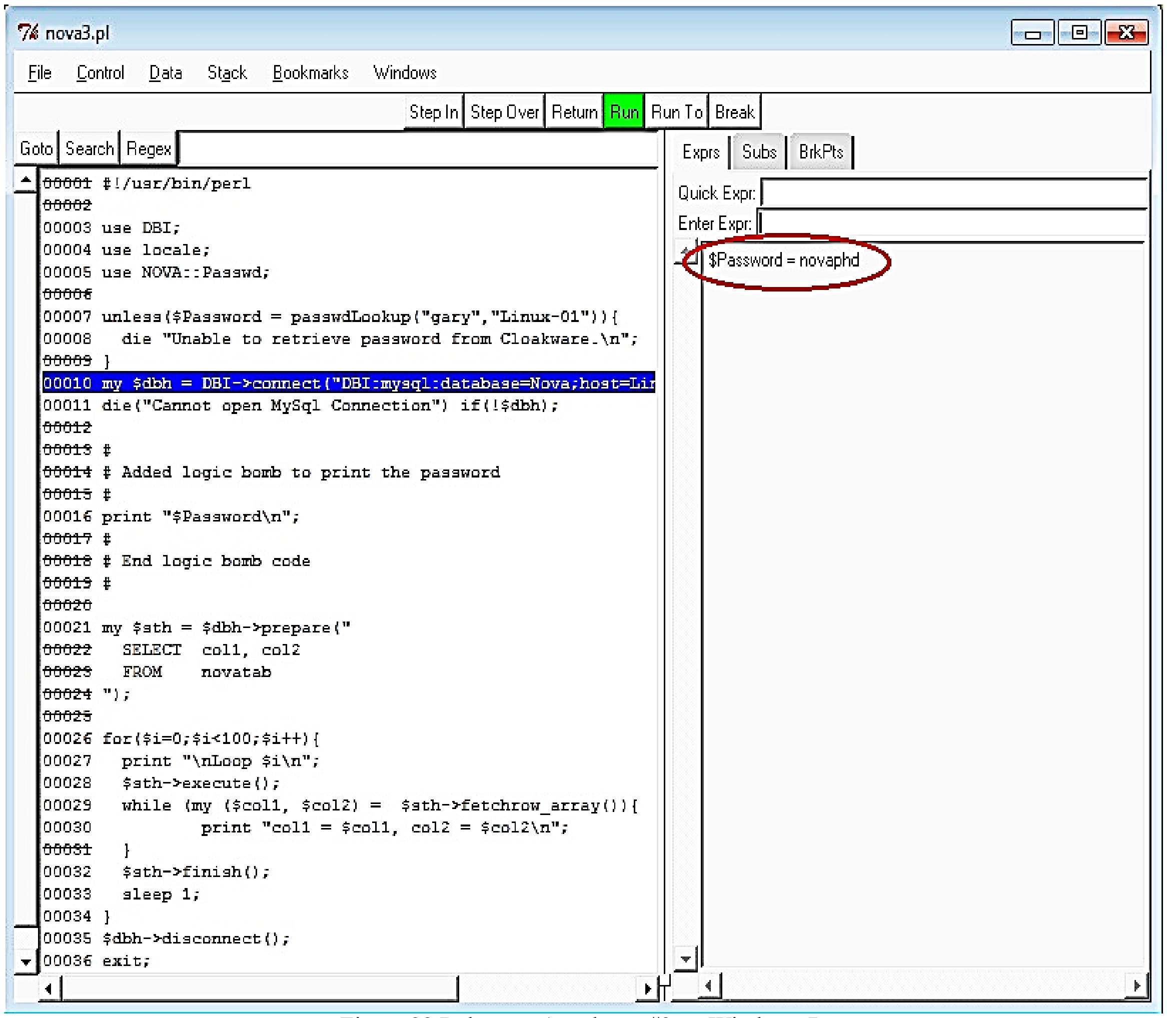
Figure 21 Debugger Attack run #1 on Linux


nova.pl
File Control Data Stack Bookmarks Windows
Step In Step Over Return Run Run To Break
Goto Search Regex
Exprs Subs BrkPts
Quick Expr:
Enter Expr:

```
#!/usr/bin/perl

use DBI;
use locale;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

Figure 22 Debugger Attack run #1 on Solaris 10

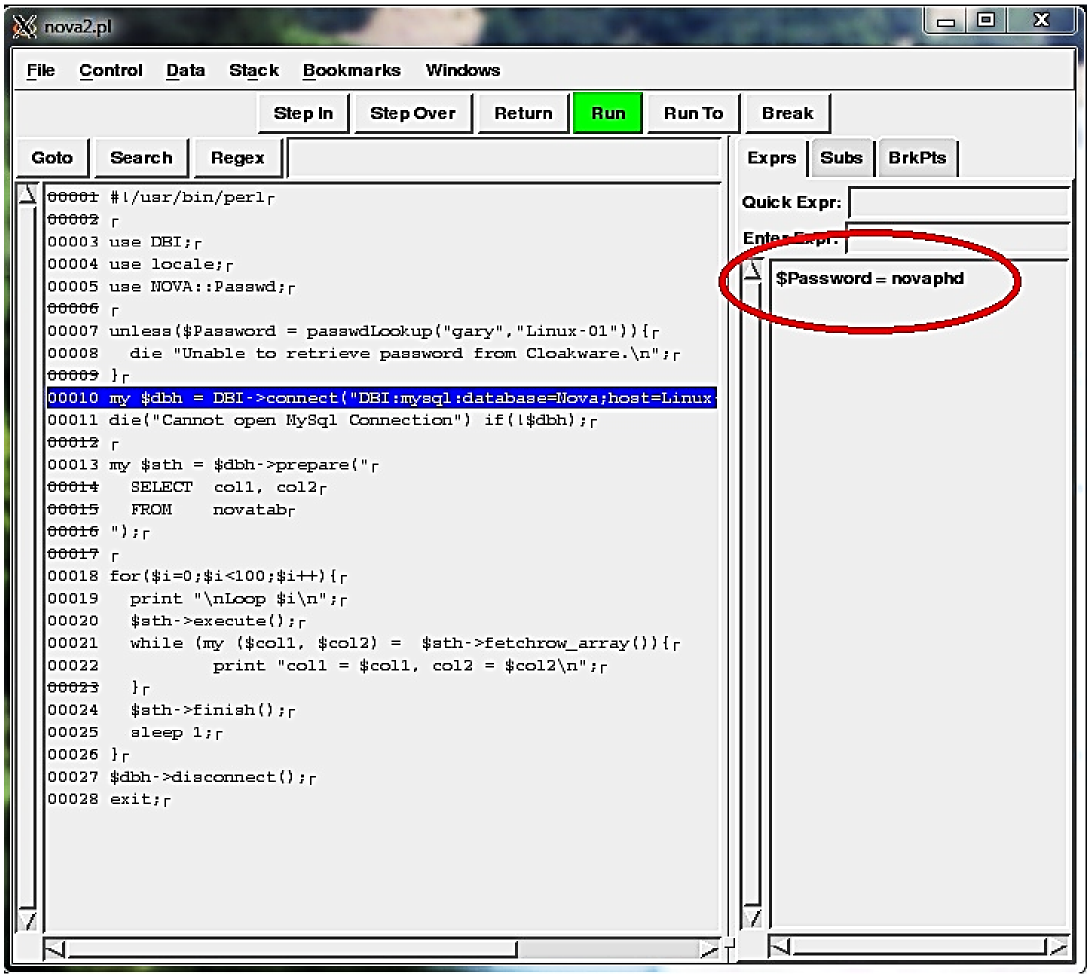
Figures 20, 21 and 22 show a successful debugger attack vector carried out on all three platforms against a simple Perl script with hard-coded credentials. The payload is highlighted in the red circle.



```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","Linux-01")){
  die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Li
die("Cannot open MySql Connection") if(!$dbh);

#
# Added logic bomb to print the password
#
print "$Password\n";
#
# End logic bomb code
#

my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

```
$Password = novaphd
```

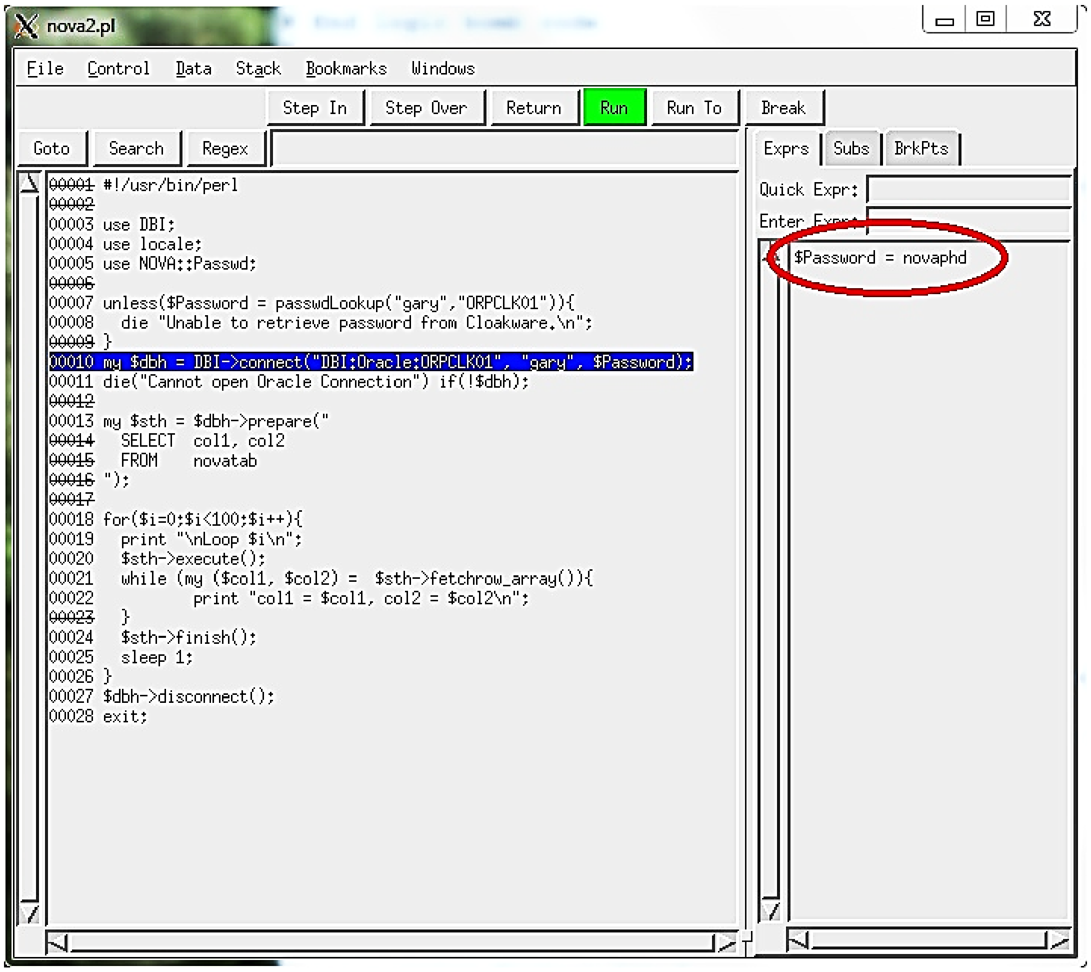
Figure 23 Debugger Attack run #2 on Windows 7



```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","Linux-01")){
  die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

```
$Password = novaphd
```

Figure 24 Debugger Attack run #2 on Linux



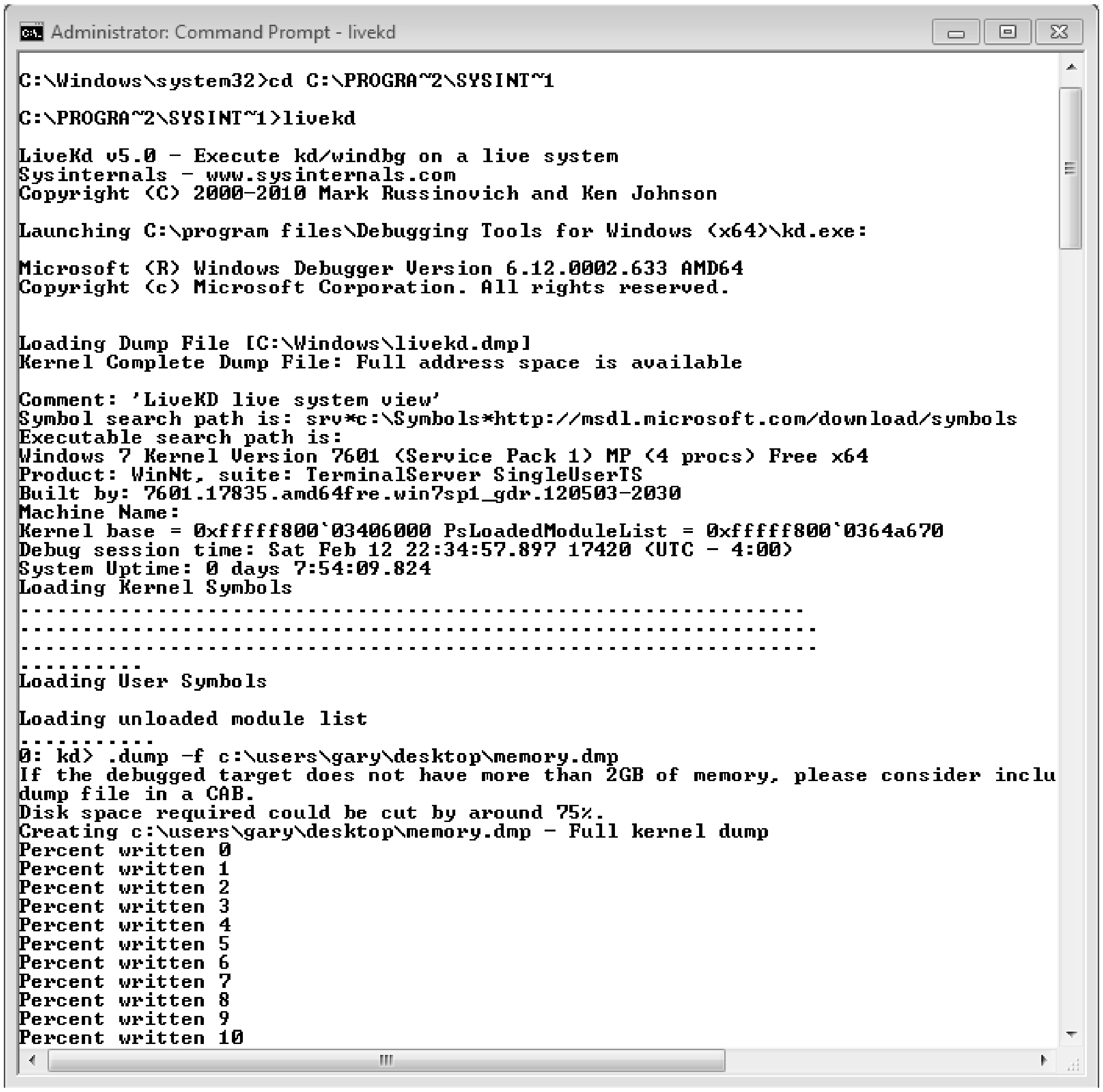
```
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

unless($Password = passwdLookup("gary","ORPCLK01")){
  die "Unable to retrieve password from Cloakware.\n";
}
my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Password);
die("Cannot open Oracle Connection") if(!$dbh);

my $sth = $dbh->prepare("
  SELECT  col1, col2
  FROM    novatab
");

for($i=0;$i<100;$i++){
  print "\nLoop $i\n";
  $sth->execute();
  while (my ($col1, $col2) =  $sth->fetchrow_array()){
          print "col1 = $col1, col2 = $col2\n";
  }
  $sth->finish();
  sleep 1;
}
$dbh->disconnect();
exit;
```

```
$Password = novaphd
```

Figure 25 Debugger Attack run #2 on Solaris 10

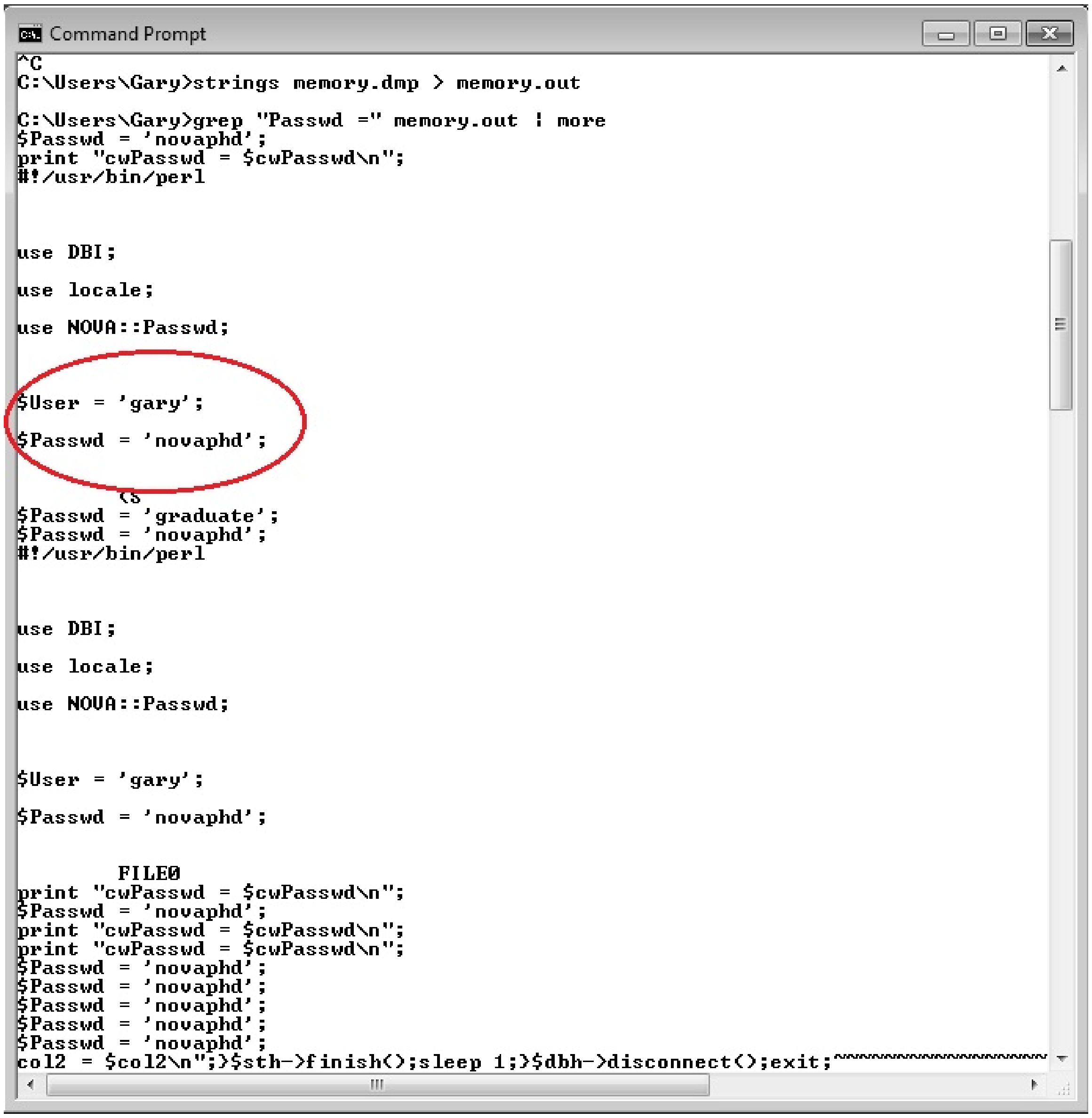
Figures 23, 24 and 25 show a successful debugger attack vector carried out on all three platforms against a simple Perl script utilizing the password vault via the passwdLookup subroutine held in the NOVA::Passwd Perl module. The payload is highlighted in the red circle.

Baseline Memory Scan Attack Vector Screenshots



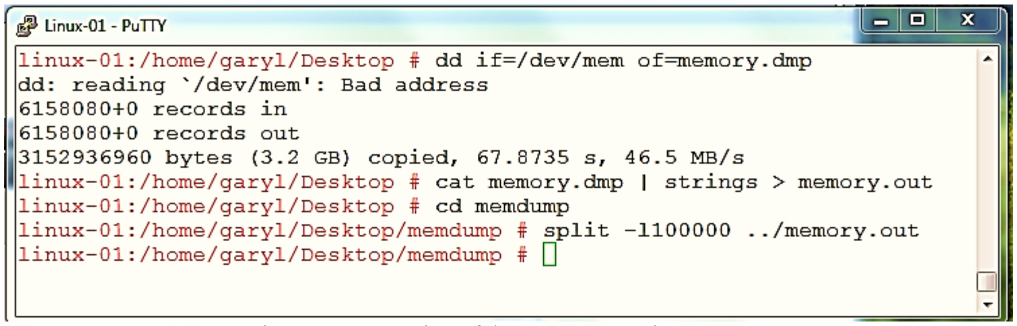
```
C:\Windows\system32>cd C:\PROGRA~2\SYSINT~1

C:\PROGRA~2\SYSINT~1>livekd

LiveKd v5.0 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2010 Mark Russinovich and Ken Johnson

Launching C:\program files\Debugging Tools for Windows (x64)\kd.exe:

Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [C:\Windows\livekd.dmp]
Kernel Complete Dump File: Full address space is available

Comment: 'LiveKD live system view'
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17835.amd64fre.win7sp1_gdr.120503-2030
Machine Name:
Kernel base = 0xfffff800`03406000 PsLoadedModuleList = 0xfffff800`0364a670
Debug session time: Sat Feb 12 22:34:57.897 17420 (UTC - 4:00)
System Uptime: 0 days 7:54:09.824
Loading Kernel Symbols
...............................................................
................................................................
................................................................
..........
Loading User Symbols

Loading unloaded module list
...........
0: kd> .dump -f c:\users\gary\desktop\memory.dmp
If the debugged target does not have more than 2GB of memory, please consider inclu
dump file in a CAB.
Disk space required could be cut by around 75%.
Creating c:\users\gary\desktop\memory.dmp - Full kernel dump
Percent written 0
Percent written 1
Percent written 2
Percent written 3
Percent written 4
Percent written 5
Percent written 6
Percent written 7
Percent written 8
Percent written 9
Percent written 10
```

Figure 26 Generation of the memory on Windows 7



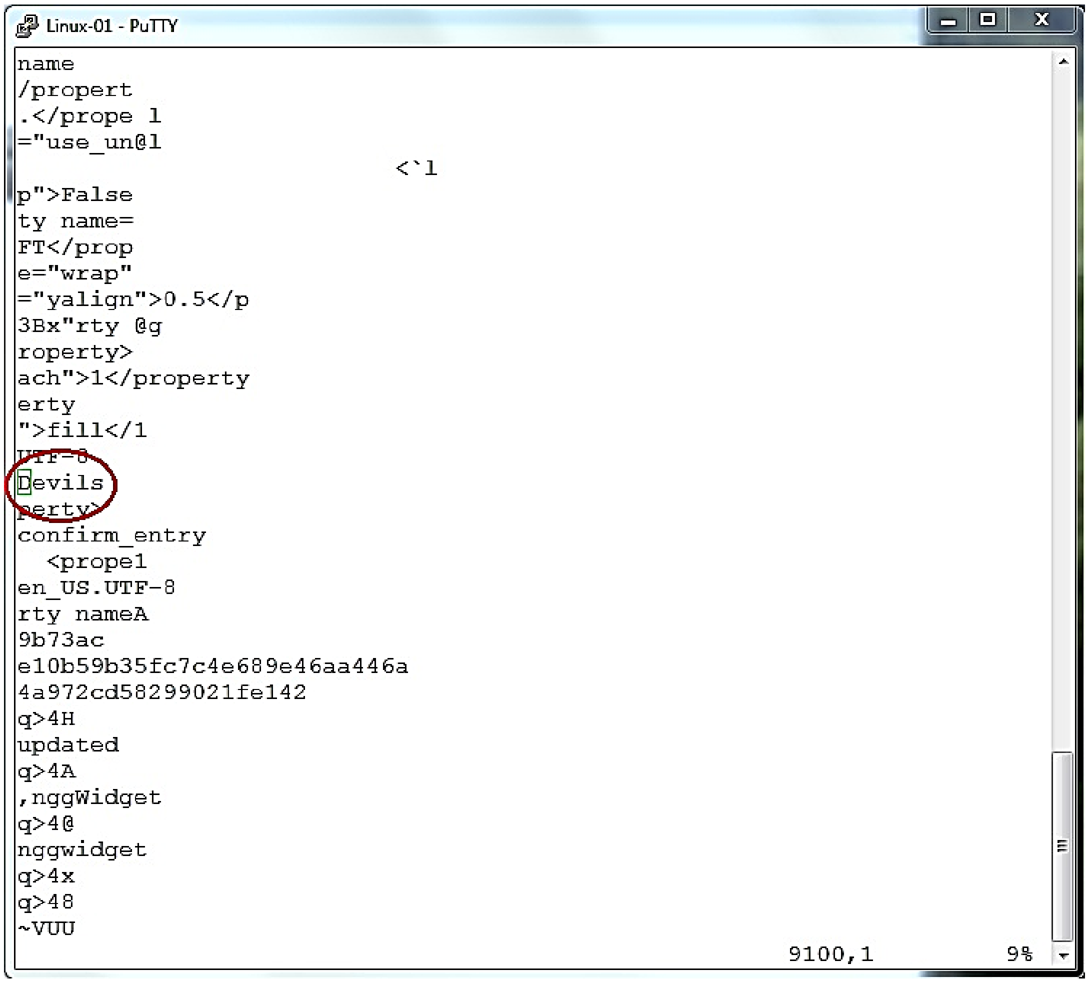
```
^C
C:\Users\Gary>strings memory.dmp > memory.out

C:\Users\Gary>grep "Passwd =" memory.out | more
$Passwd = 'novaphd';
print "cwPasswd = $cwPasswd\n";
#!/usr/bin/perl


use DBI;

use locale;

use NOVA::Passwd;


$User = 'gary';

$Passwd = 'novaphd';

        (s
$Passwd = 'graduate';
$Passwd = 'novaphd';
#!/usr/bin/perl


use DBI;

use locale;

use NOVA::Passwd;


$User = 'gary';

$Passwd = 'novaphd';


        FILE0
print "cwPasswd = $cwPasswd\n";
$Passwd = 'novaphd';
print "cwPasswd = $cwPasswd\n";
print "cwPasswd = $cwPasswd\n";
$Passwd = 'novaphd';
$Passwd = 'novaphd';
$Passwd = 'novaphd';
$Passwd = 'novaphd';
$Passwd = 'novaphd';
col2 = $col2\n";}$sth->finish();sleep 1;}$dbh->disconnect();exit;
```

Figure 27 Memory scan attack vector #1 on Windows 7



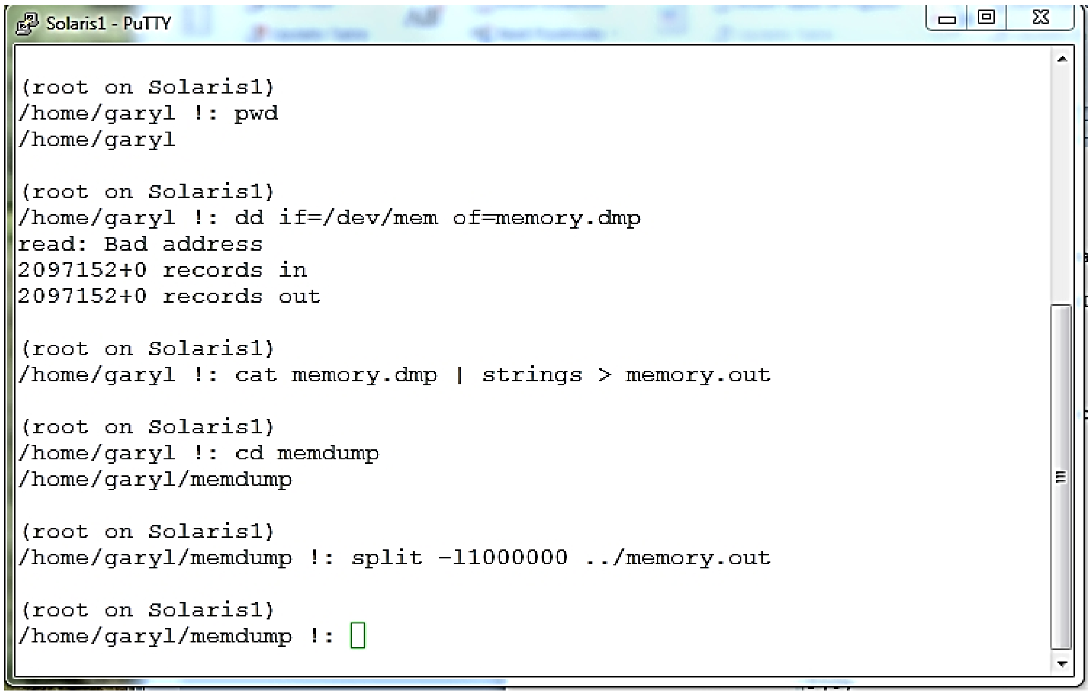
```
linux-01:/home/garyl/Desktop # dd if=/dev/mem of=memory.dmp
dd: reading `/dev/mem': Bad address
6158080+0 records in
6158080+0 records out
3152936960 bytes (3.2 GB) copied, 67.8735 s, 46.5 MB/s
linux-01:/home/garyl/Desktop # cat memory.dmp | strings > memory.out
linux-01:/home/garyl/Desktop # cd memdump
linux-01:/home/garyl/Desktop/memdump # split -l100000 ../memory.out
linux-01:/home/garyl/Desktop/memdump #
```

Figure 28 Generation of the memory on Linux



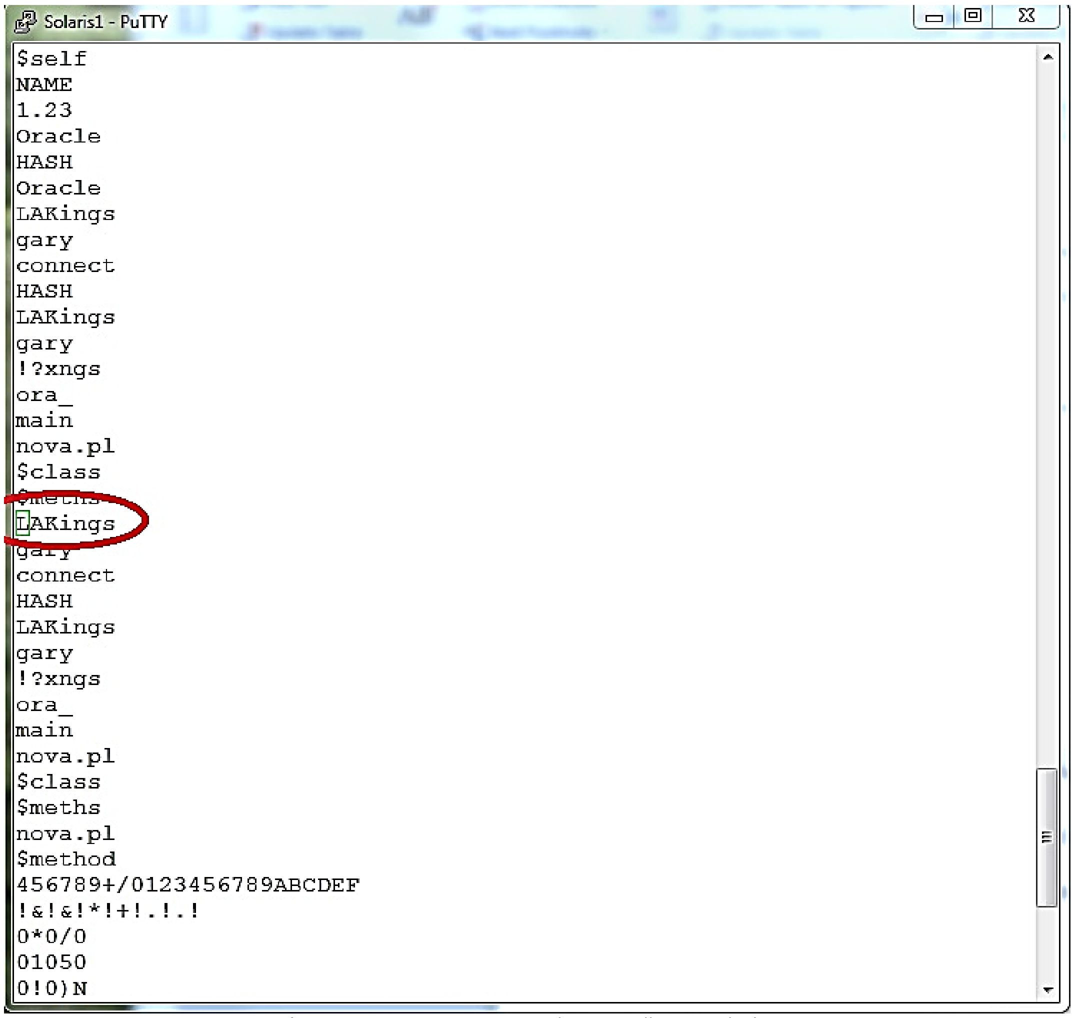
```
name
/propert
.</prope 1
="use_un@l
                          <`l
p">False
ty name=
FT</prop
e="wrap"
="yalign">0.5</p
3Bx"rty @g
roperty>
ach">1</property
erty
">fill</1
UTF-8
Devils
perty>
confirm_entry
  <prope1
en_US.UTF-8
rty nameA
9b73ac
e10b59b35fc7c4e689e46aa446a
4a972cd58299021fe142
q>4H
updated
q>4A
,nggWidget
q>4@
nggwidget
q>4x
q>48
~VUU
                                                             9100,1        9%
```

Figure 29 Memory scan attack vector #1 on Linux



```
(root on Solaris1)
/home/garyl !: pwd
/home/garyl

(root on Solaris1)
/home/garyl !: dd if=/dev/mem of=memory.dmp
read: Bad address
2097152+0 records in
2097152+0 records out

(root on Solaris1)
/home/garyl !: cat memory.dmp | strings > memory.out

(root on Solaris1)
/home/garyl !: cd memdump
/home/garyl/memdump

(root on Solaris1)
/home/garyl/memdump !: split -l1000000 ../memory.out

(root on Solaris1)
/home/garyl/memdump !:
```

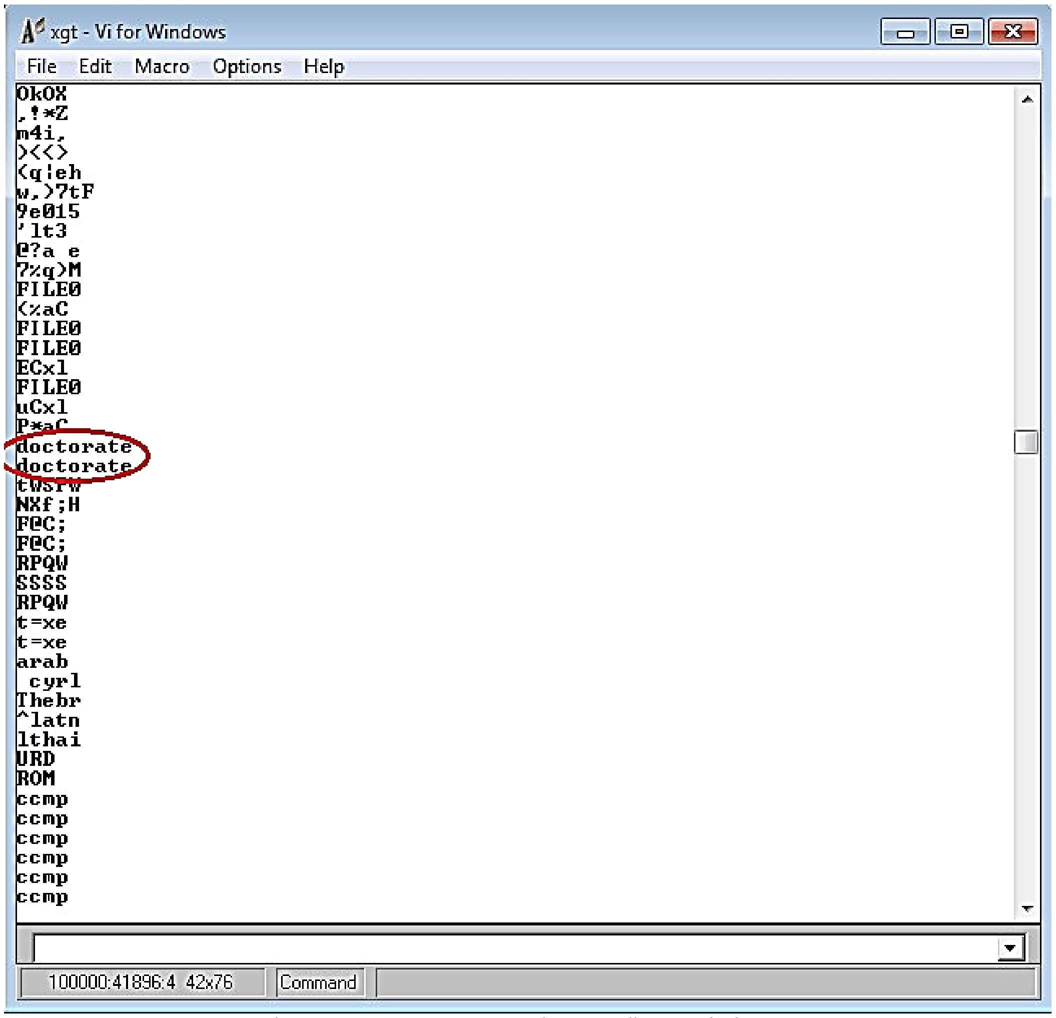
Figure 30 Generation of the memory on Solaris 10



```
$self
NAME
1.23
Oracle
HASH
Oracle
LAKings
gary
connect
HASH
LAKings
gary
!?xngs
ora_
main
nova.pl
$class
$meths
LAKings
gary
connect
HASH
LAKings
gary
!?xngs
ora_
main
nova.pl
$class
$meths
nova.pl
$method
456789+/0123456789ABCDEF
!&!&!*!+!.!.!
0*0/0
01050
0!0)N
```

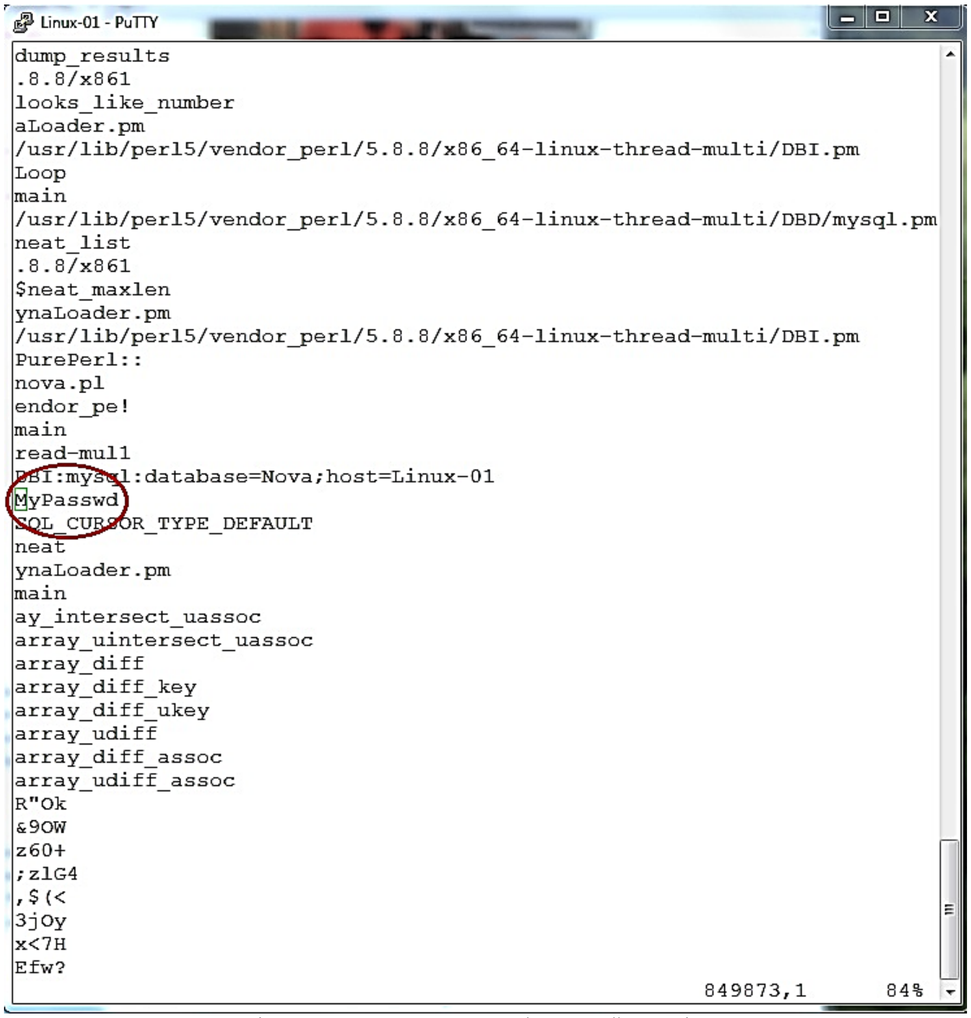
Figure 31 Memory scan attack vector #1 on Solaris 10

Figures 26 through 31 show a successful memory scan attack vector carried out on all three platforms against a simple Perl script with hard-coded credentials. The payload is highlighted in the red circles. Figures 26, 28 and 30 show the generation of the memory dumps.



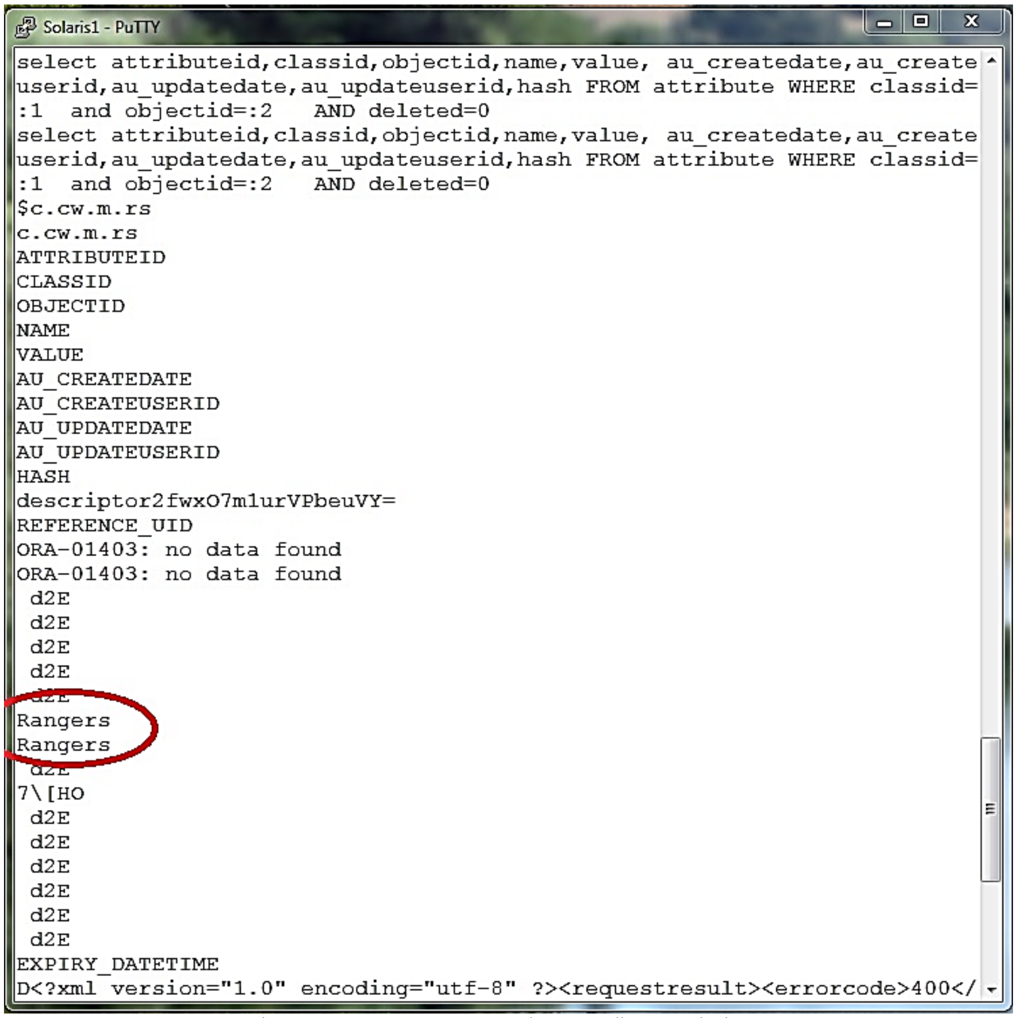
```
OkOX
,!*Z
m4i,
><<>
<q!eh
w,>7tF
9e015
'lt3
@?a e
?%q>M
FILE0
<%aC
FILE0
FILE0
ECxl
FILE0
uCxl
P*aC
doctorate
doctorate
twsrw
NXf;H
F@C;
F@C;
RPQW
SSSS
RPQW
t=xe
t=xe
arab
 cyrl
Thebr
^latn
lthai
URD
ROM
ccmp
ccmp
ccmp
ccmp
ccmp
ccmp
```

Figure 32 Memory scan attack vector #2 on Windows 7



```
dump_results
.8.8/x861
looks_like_number
aLoader.pm
/usr/lib/perl5/vendor_perl/5.8.8/x86_64-linux-thread-multi/DBI.pm
Loop
main
/usr/lib/perl5/vendor_perl/5.8.8/x86_64-linux-thread-multi/DBD/mysql.pm
neat_list
.8.8/x861
$neat_maxlen
ynaLoader.pm
/usr/lib/perl5/vendor_perl/5.8.8/x86_64-linux-thread-multi/DBI.pm
PurePerl::
nova.pl
endor_pe!
main
read-mul1
DBI:mysql:database=Nova;host=Linux-01
MyPasswd
SQL_CURSOR_TYPE_DEFAULT
neat
ynaLoader.pm
main
ay_intersect_uassoc
array_uintersect_uassoc
array_diff
array_diff_key
array_diff_ukey
array_udiff
array_diff_assoc
array_udiff_assoc
R"Ok
&9OW
z60+
;zlG4
,$(<
3jOy
x<7H
Efw?
                                                  849873,1      84%
```

Figure 33 Memory scan attack vector #2 on Linux

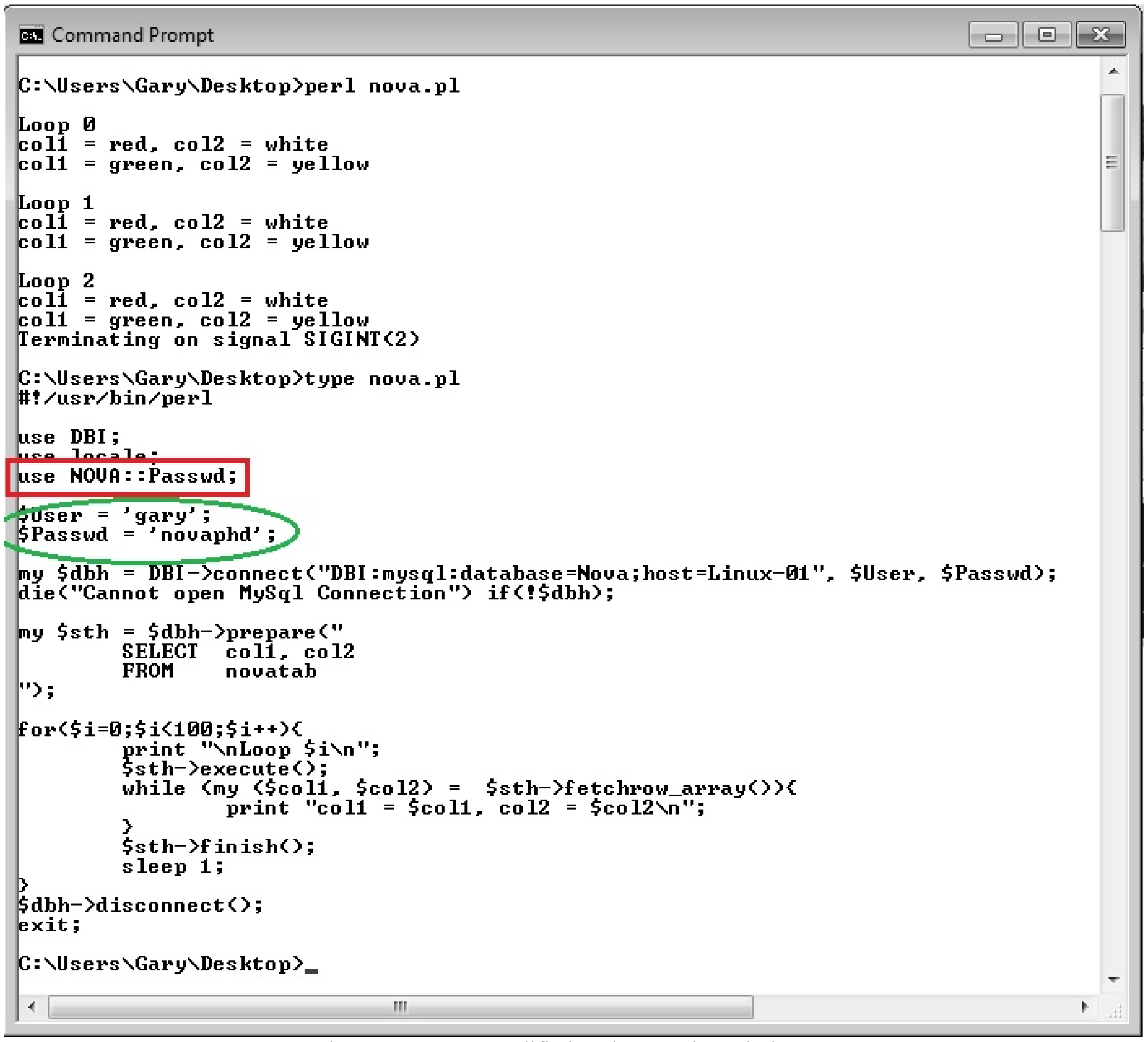


```
select attributeid,classid,objectid,name,value, au_createdate,au_create
userid,au_updatedate,au_updateuserid,hash FROM attribute WHERE classid=
:1  and objectid=:2   AND deleted=0
select attributeid,classid,objectid,name,value, au_createdate,au_create
userid,au_updatedate,au_updateuserid,hash FROM attribute WHERE classid=
:1  and objectid=:2   AND deleted=0
$c.cw.m.rs
c.cw.m.rs
ATTRIBUTEID
CLASSID
OBJECTID
NAME
VALUE
AU_CREATEDATE
AU_CREATEUSERID
AU_UPDATEDATE
AU_UPDATEUSERID
HASH
descriptor2fwxO7m1urVPbeuVY=
REFERENCE_UID
ORA-01403: no data found
ORA-01403: no data found
 d2E
 d2E
 d2E
 d2E
 d2E
Rangers
Rangers
 d2E
7\[HO
 d2E
 d2E
 d2E
 d2E
 d2E
 d2E
EXPIRY_DATETIME
D<?xml version="1.0" encoding="utf-8" ?><requestresult><errorcode>400</
```

Figure 34 Memory scan attack vector #2 on Solaris 10

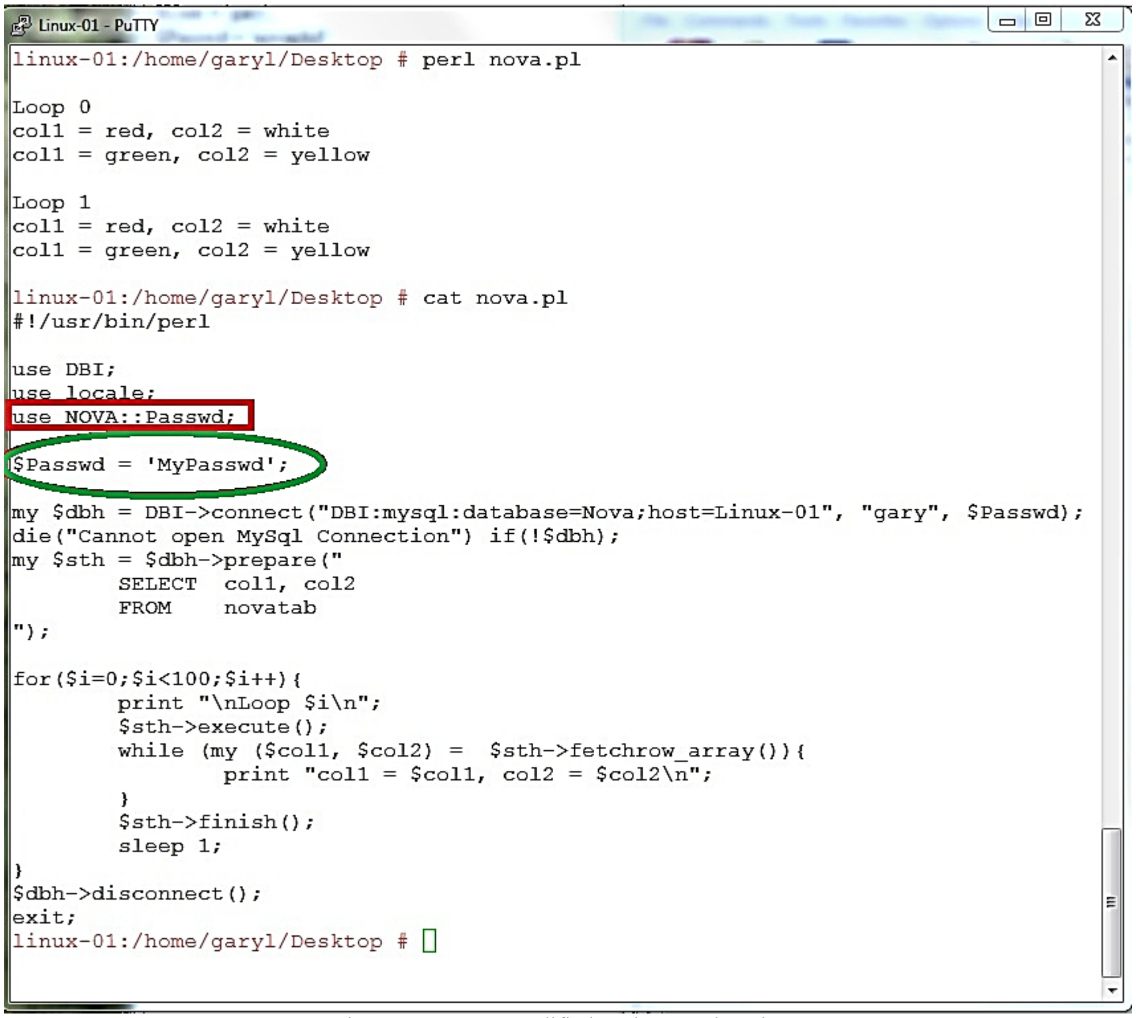
Figures 32, 33 and 34 show a successful memory scan attack vector carried out on all three platforms against a simple Perl script utilizing the password vault via the passwdLookup subroutine held in the NOVA::Passwd Perl module. The payload is highlighted in the red circle.

Aspect Modified Probe Attack Vector Screenshots



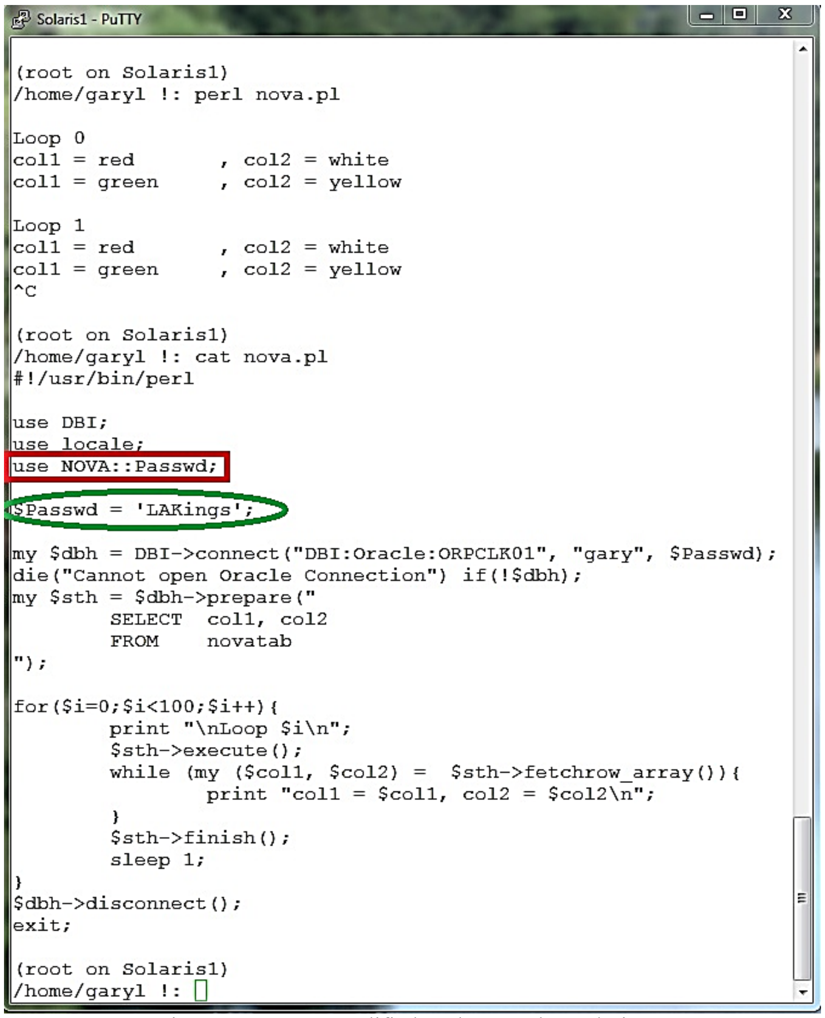
```
C:\Users\Gary\Desktop>perl nova.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 2
col1 = red, col2 = white
col1 = green, col2 = yellow
Terminating on signal SIGINT(2)

C:\Users\Gary\Desktop>type nova.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User, $Passwd);
die("Cannot open MySql Connection") if(!$dbh);

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

C:\Users\Gary\Desktop>_
```

Figure 35 Aspect Modified Probe Attack - Windows 7



```
linux-01:/home/garyl/Desktop # perl nova.pl

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

linux-01:/home/garyl/Desktop # cat nova.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$Passwd = 'MyPasswd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Passwd);
die("Cannot open MySql Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
linux-01:/home/garyl/Desktop #
```

Figure 36 Aspect Modified Probe Attack - Linux



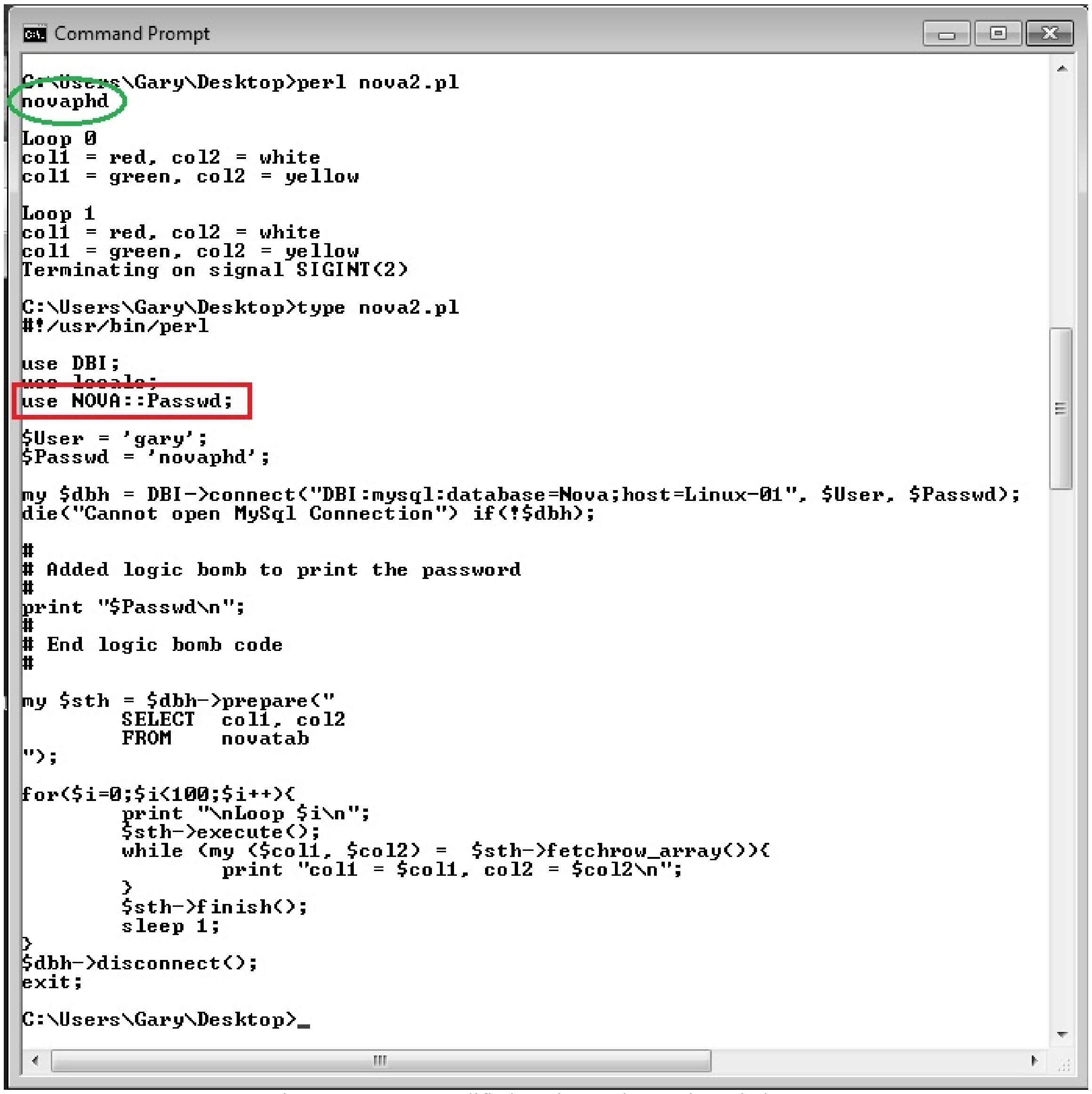
```
(root on Solaris1)
/home/garyl !: perl nova.pl

Loop 0
col1 = red       , col2 = white
col1 = green     , col2 = yellow

Loop 1
col1 = red       , col2 = white
col1 = green     , col2 = yellow
^C

(root on Solaris1)
/home/garyl !: cat nova.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$Passwd = 'LAKings';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

(root on Solaris1)
/home/garyl !: 
```

Figure 37 Aspect Modified Probe Attack - Solaris 10

Figures 35, 36 and 37 demonstrate a failed probe attack vector on the Windows 7, Linux ad Solaris 10 platforms. The script avoids attack by activating and firing an aspect through the 'use NOVA::Passwd;' pragma (highlighted in the green circle) and obfuscates the real password by setting (highlighted in the red rectangle) and passing a phony password to the DBI->connect subroutine.

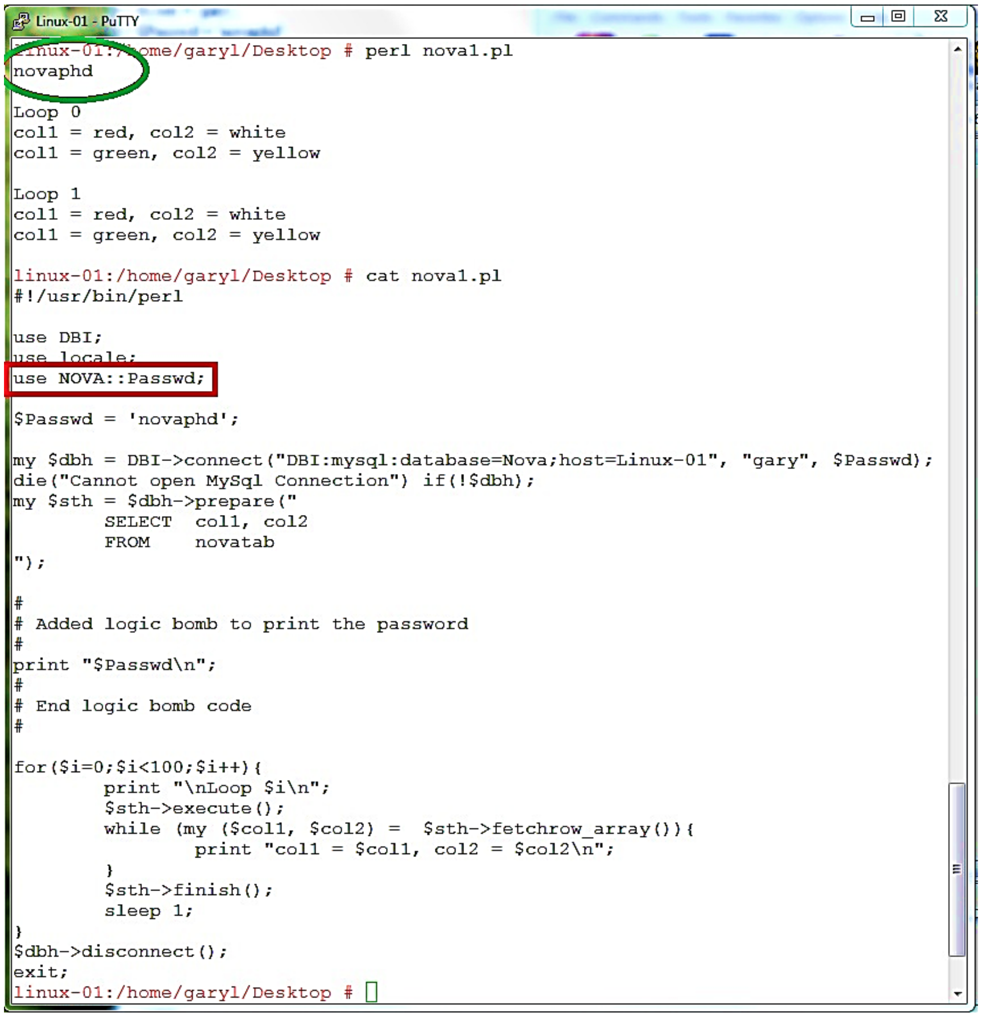
Aspect Modified Logic Bomb Attack Vector Screenshots

Figures 38, 39 and 40 demonstrate a failed logic bomb attack vector on the Windows 7, Linux ad Solaris 10 platforms. The script avoids attack by activating and firing an aspect through the ‘use NOVA::Passwd;’ pragma and obfuscates the real password by setting (highlighted in the red rectangle) and passing a phony password to the DBI->connect subroutine.   The attack payload is shown in the green circle and is broadcasting the wrong password.



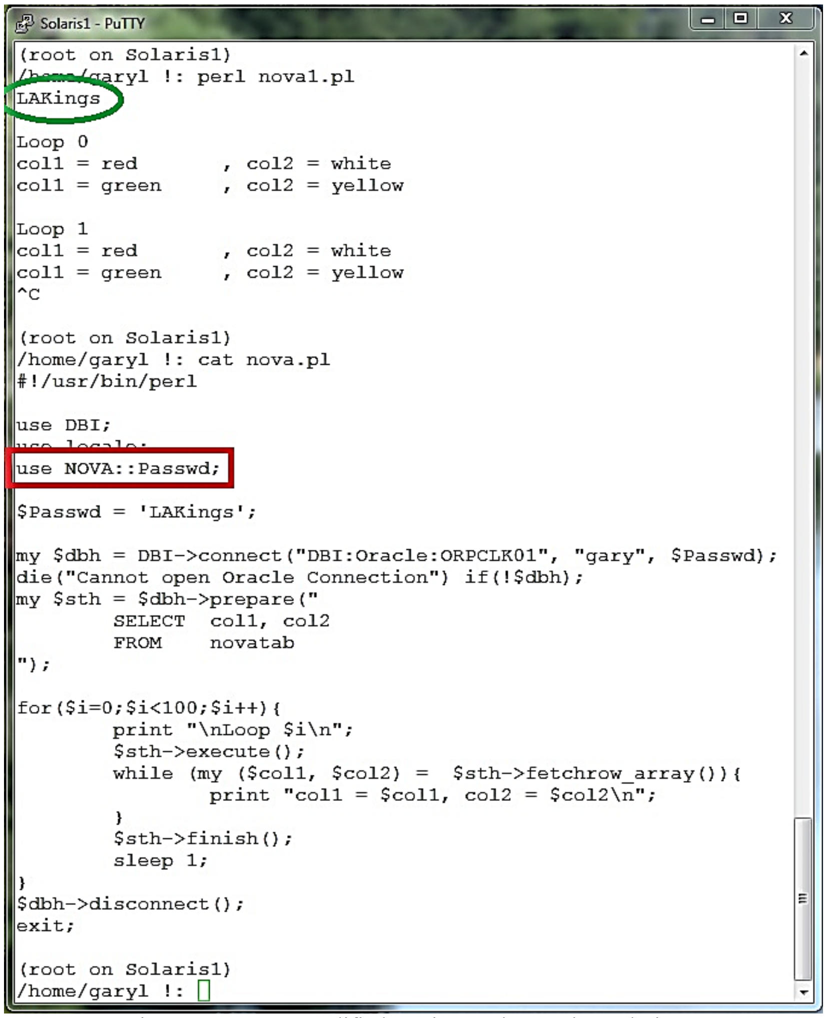
```
C:\Users\Gary\Desktop>perl nova2.pl
novaphd

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow
Terminating on signal SIGINT(2)

C:\Users\Gary\Desktop>type nova2.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$User = 'gary';
$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", $User, $Passwd);
die("Cannot open MySql Connection") if(!$dbh);

#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#

my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

C:\Users\Gary\Desktop>_
```

Figure 38 Aspect Modified Logic Bomb Attack - Windows 7



```
linux-01:/home/garyl/Desktop # perl nova1.pl
novaphd

Loop 0
col1 = red, col2 = white
col1 = green, col2 = yellow

Loop 1
col1 = red, col2 = white
col1 = green, col2 = yellow

linux-01:/home/garyl/Desktop # cat nova1.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$Passwd = 'novaphd';

my $dbh = DBI->connect("DBI:mysql:database=Nova;host=Linux-01", "gary", $Passwd);
die("Cannot open MySql Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

#
# Added logic bomb to print the password
#
print "$Passwd\n";
#
# End logic bomb code
#

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;
linux-01:/home/garyl/Desktop #
```

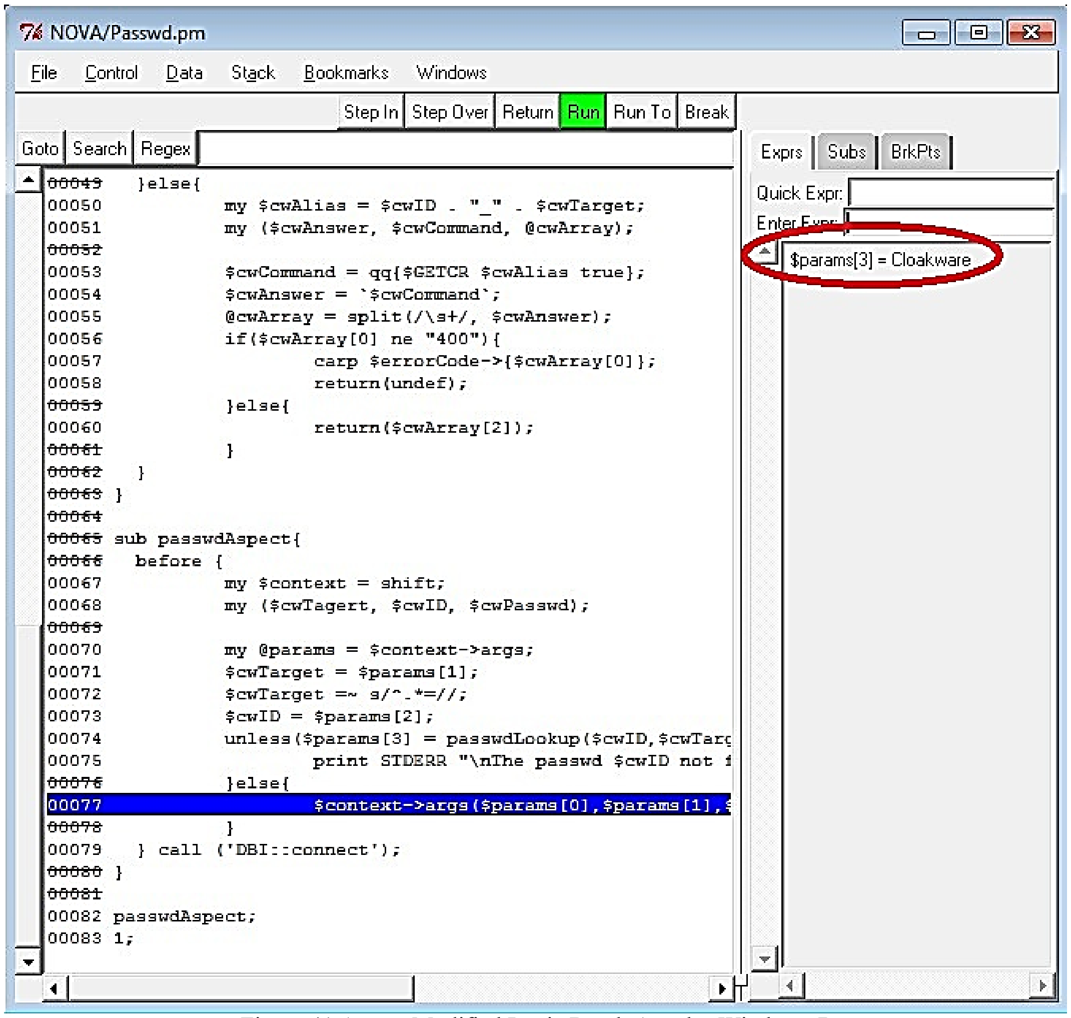
Figure 39 Aspect Modified Logic Bomb Attack - Linux



```
(root on Solaris1)
/home/garyl !: perl nova1.pl
LAKings

Loop 0
col1 = red       , col2 = white
col1 = green     , col2 = yellow

Loop 1
col1 = red       , col2 = white
col1 = green     , col2 = yellow
^C

(root on Solaris1)
/home/garyl !: cat nova.pl
#!/usr/bin/perl

use DBI;
use locale;
use NOVA::Passwd;

$Passwd = 'LAKings';

my $dbh = DBI->connect("DBI:Oracle:ORPCLK01", "gary", $Passwd);
die("Cannot open Oracle Connection") if(!$dbh);
my $sth = $dbh->prepare("
        SELECT  col1, col2
        FROM    novatab
");

for($i=0;$i<100;$i++){
        print "\nLoop $i\n";
        $sth->execute();
        while (my ($col1, $col2) =  $sth->fetchrow_array()){
                print "col1 = $col1, col2 = $col2\n";
        }
        $sth->finish();
        sleep 1;
}
$dbh->disconnect();
exit;

(root on Solaris1)
/home/garyl !:
```

Figure 40 Aspect Modified Logic Bomb Attack - Solaris 10

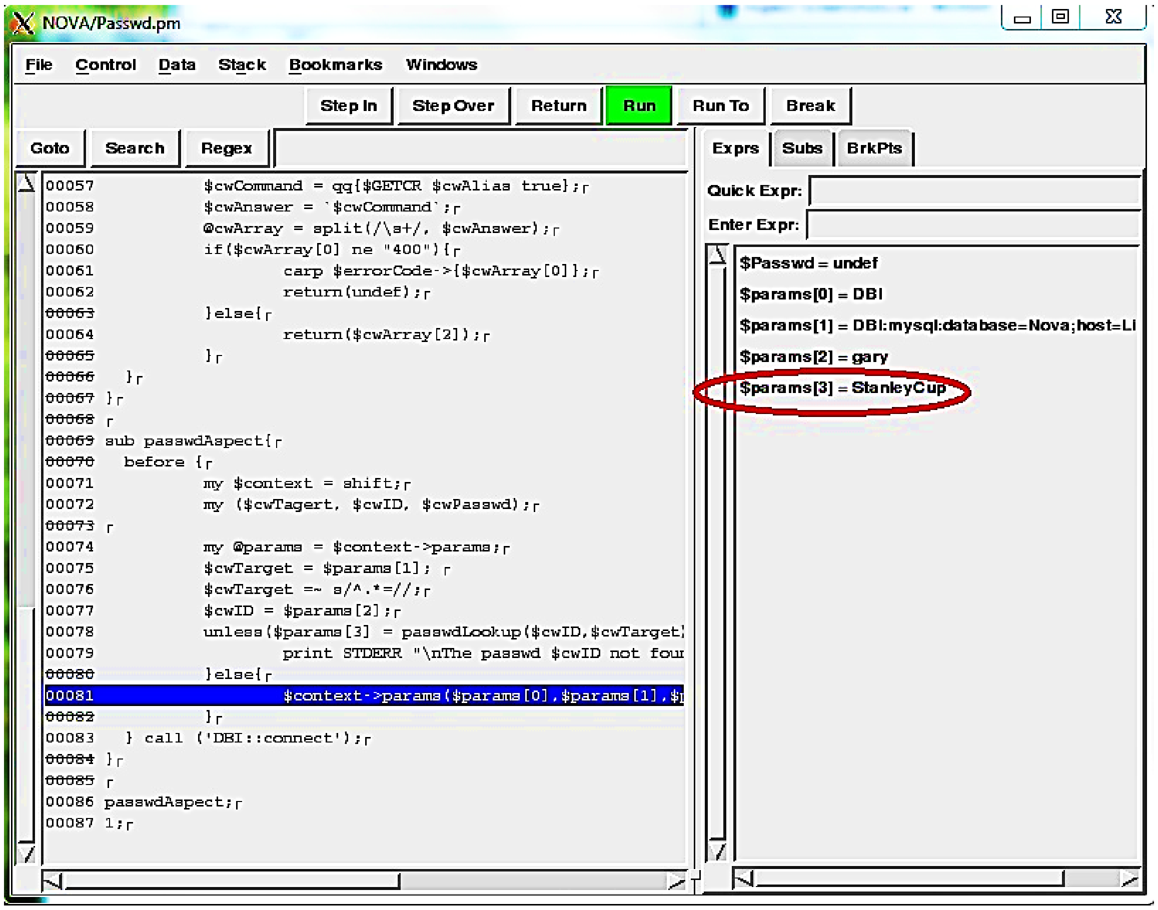
Aspect Modified Debugger Attack Vector Screenshots



```
    }else{
            my $cwAlias = $cwID . "_" . $cwTarget;
            my ($cwAnswer, $cwCommand, @cwArray);

            $cwCommand = qq{$GETCR $cwAlias true};
            $cwAnswer = `$cwCommand`;
            @cwArray = split(/\s+/, $cwAnswer);
            if($cwArray[0] ne "400"){
                    carp $errorCode->{$cwArray[0]};
                    return(undef);
            }else{
                    return($cwArray[2]);
            }
    }
}

sub passwdAspect{
  before {
            my $context = shift;
            my ($cwTagert, $cwID, $cwPasswd);

            my @params = $context->args;
            $cwTarget = $params[1];
            $cwTarget =~ s/^.*=//;
            $cwID = $params[2];
            unless($params[3] = passwdLookup($cwID,$cwTar
                    print STDERR "\nThe passwd $cwID not
            }else{
                    $context->args($params[0],$params[1],
            }
  } call ('DBI::connect');
}

passwdAspect;
1;
```

$params[3] = Cloakware

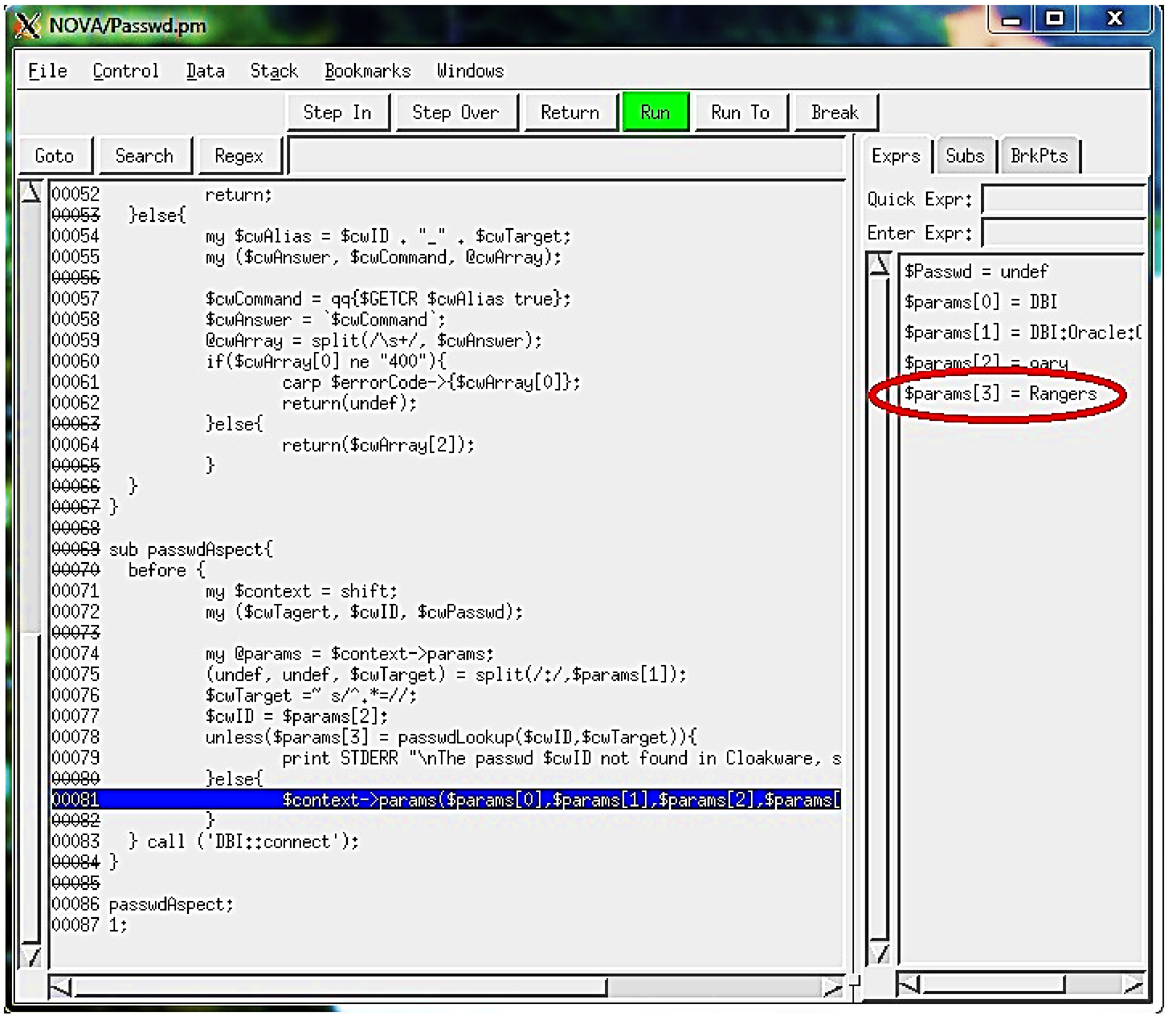
Figure 41 Aspect Modified Logic Bomb Attack - Windows 7



```
          $cwCommand = qq{$GETCR $cwAlias true};┌
          $cwAnswer = `$cwCommand`;┌
          @cwArray = split(/\s+/, $cwAnswer);┌
          if($cwArray[0] ne "400"){┌
                  carp $errorCode->{$cwArray[0]};┌
                  return(undef);┌
          }else{┌
                  return($cwArray[2]);┌
          }┌
  }┌
}┌
┌
sub passwdAspect{┌
  before {┌
          my $context = shift;┌
          my ($cwTagert, $cwID, $cwPasswd);┌
┌
          my @params = $context->params;┌
          $cwTarget = $params[1]; ┌
          $cwTarget =~ s/^.*=//;┌
          $cwID = $params[2];┌
          unless($params[3] = passwdLookup($cwID,$cwTarget)
                  print STDERR "\nThe passwd $cwID not fou
          }else{┌
                  $context->params($params[0],$params[1],$p
          }┌
  } call ('DBI::connect');┌
}┌
┌
passwdAspect;┌
1;┌
```

```
$Passwd = undef
$params[0] = DBI
$params[1] = DBI:mysql:database=Nova;host=Li
$params[2] = gary
$params[3] = StanleyCup
```

Figure 42 Aspect Modified Logic Bomb Attack – Linux



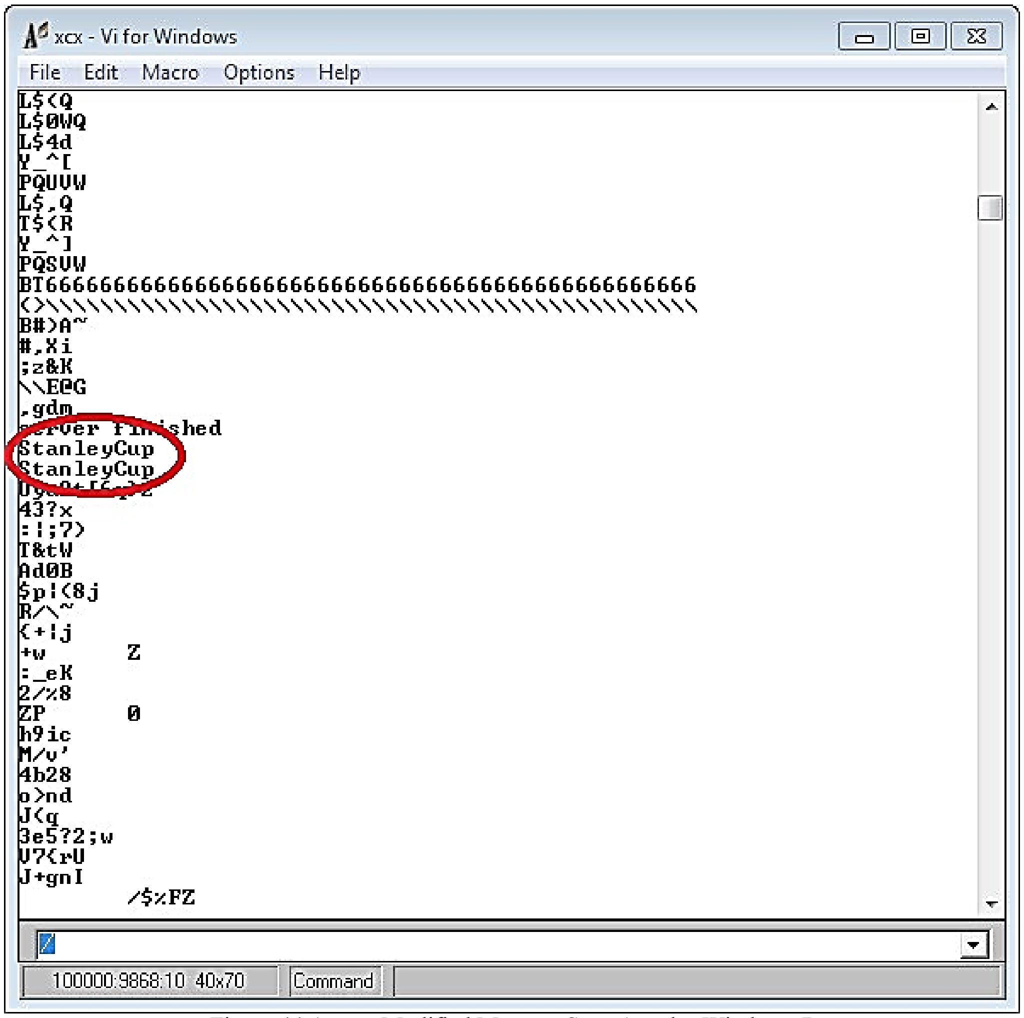
```
                return;
        }else{
                my $cwAlias = $cwID . "_" . $cwTarget;
                my ($cwAnswer, $cwCommand, @cwArray);

                $cwCommand = qq{$GETCR $cwAlias true};
                $cwAnswer = `$cwCommand`;
                @cwArray = split(/\s+/, $cwAnswer);
                if($cwArray[0] ne "400"){
                        carp $errorCode->{$cwArray[0]};
                        return(undef);
                }else{
                        return($cwArray[2]);
                }
        }
}

sub passwdAspect{
  before {
                my $context = shift;
                my ($cwTagert, $cwID, $cwPasswd);

                my @params = $context->params;
                (undef, undef, $cwTarget) = split(/:/,$params[1]);
                $cwTarget =~ s/^.*=//;
                $cwID = $params[2];
                unless($params[3] = passwdLookup($cwID,$cwTarget)){
                        print STDERR "\nThe passwd $cwID not found in Cloakware, s
                }else{
                        $context->params($params[0],$params[1],$params[2],$params[
                }
  } call ('DBI::connect');
}

passwdAspect;
1;
```

```
$Passwd = undef
$params[0] = DBI
$params[1] = DBI:Oracle:(
$params[2] = gary
$params[3] = Rangers
```

Figure 43 Aspect Modified Logic Bomb Attack - Solaris 10

Figures 41, 42 and 43 show a successful debugger attack vector run against all three platforms. The payload is circled in red. On average this attack took 50 debugger steps to accomplish.

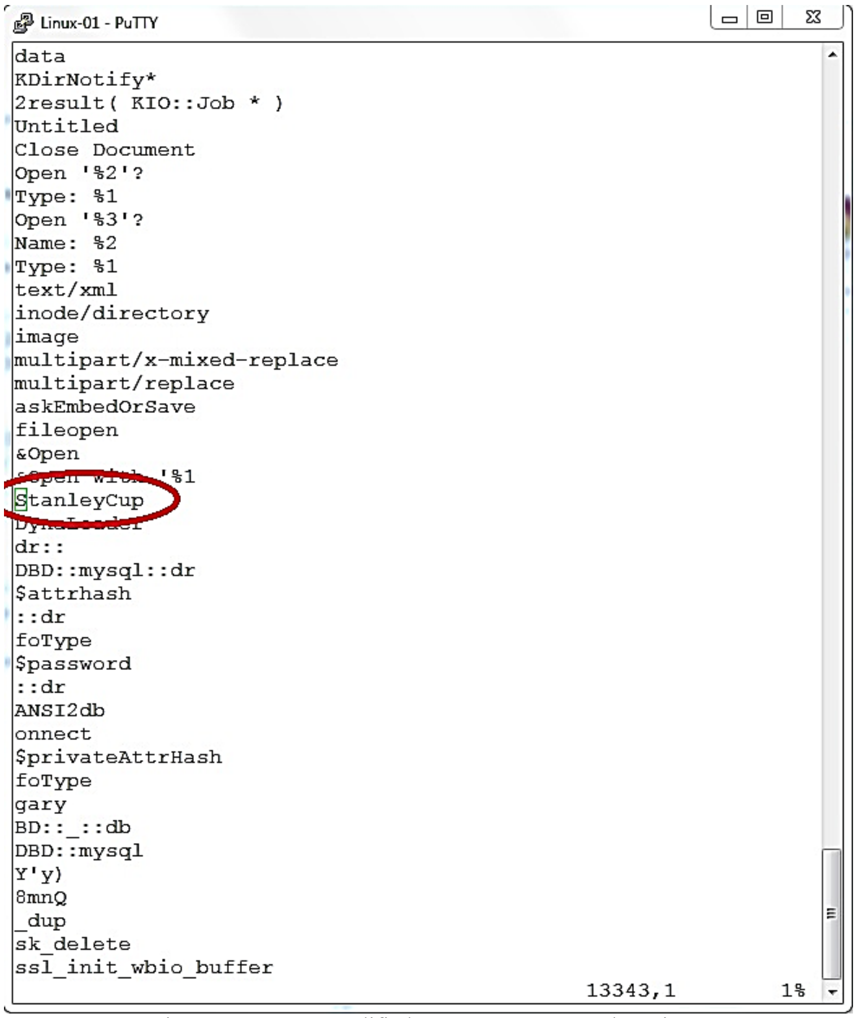
Aspect Modified Memory Scan Attack Vector Screenshots



```
L$<Q
L$0WQ
L$4d
Y_^[
PQUVW
L$,Q
T$<R
Y_^]
PQSUW
BT666666666666666666666666666666666666666666666666
<>\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
B#>A~
#,Xi
;z&K
\\E@G
,gdm
server finished
StanleyCup
StanleyCup
Uy[illegible]Z
43?x
:|;7>
T&tW
Ad0B
$p|<8j
R/\~
<+|j
+w       Z
:_eK
2/%8
ZP       0
h9ic
M/v'
4b28
o>nd
J<q
3e5?2;w
V7<rU
J+gnI
        /$%FZ
```

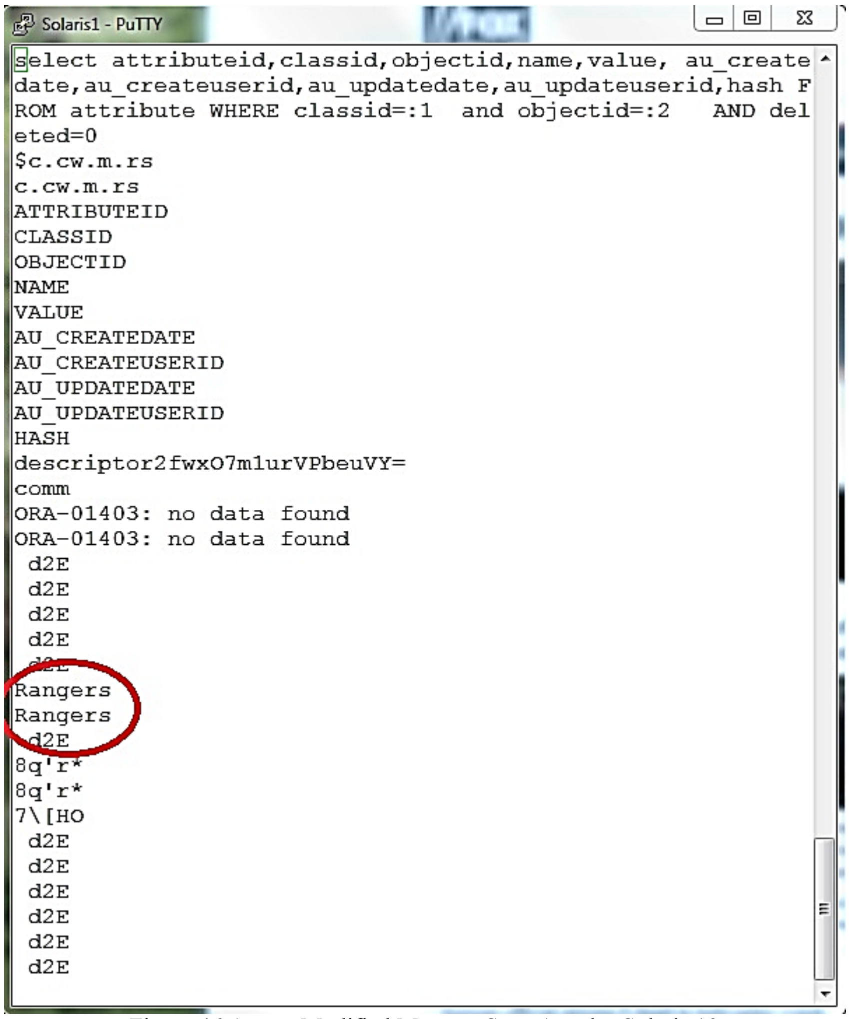
```
100000:9868:10  40x70    Command
```

Figure 44 Aspect Modified Memory Scan Attack - Windows 7



```
data
KDirNotify*
2result( KIO::Job * )
Untitled
Close Document
Open '%2'?
Type: %1
Open '%3'?
Name: %2
Type: %1
text/xml
inode/directory
image
multipart/x-mixed-replace
multipart/replace
askEmbedOrSave
fileopen
&Open
&Open with '%1
StanleyCup
DynaLoader
dr::
DBD::mysql::dr
$attrhash
::dr
foType
$password
::dr
ANSI2db
onnect
$privateAttrHash
foType
gary
BD::_::db
DBD::mysql
Y'y)
8mnQ
_dup
sk_delete
ssl_init_wbio_buffer
                                             13343,1         1%
```

Figure 45 Aspect Modified Memory Scan Attack – Linux



```
select attributeid,classid,objectid,name,value, au_create
date,au_createuserid,au_updatedate,au_updateuserid,hash F
ROM attribute WHERE classid=:1  and objectid=:2   AND del
eted=0
$c.cw.m.rs
c.cw.m.rs
ATTRIBUTEID
CLASSID
OBJECTID
NAME
VALUE
AU_CREATEDATE
AU_CREATEUSERID
AU_UPDATEDATE
AU_UPDATEUSERID
HASH
descriptor2fwxO7m1urVPbeuVY=
comm
ORA-01403: no data found
ORA-01403: no data found
 d2E
 d2E
 d2E
 d2E
 d2E
Rangers
Rangers
 d2E
8q'r*
8q'r*
7\[HO
 d2E
 d2E
 d2E
 d2E
 d2E
 d2E
```

Figure 46 Aspect Modified Memory Scan Attack - Solaris 10

Figures 44, 45 and 46 show a successful memory scan attack run against all three platforms.

References

2010 Financial Services Global Security Study. (2010). Deloitte Touche Tohmatsu.

2012 Data Breach Investigations Report. (2012). from http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf

Adaikkappan, A. (2009). Application Security Controls: An Audit Perspective. *Journal Online, 6*.

Adams, G. D., Grapes, R., Gu, Y. X., Mehan, R. E. J., & Rong, J. J. (2008).

Application Identity Management Implementation Guide. (2009). Retrieved from www.cyberark.com

Application Password Management Module. (2009). Retrieved from http://www.e-dmzsecurity.com/tpam_brochures.html

Bauer, M. (2009). Anthony Lineberry on /dev/mem rootkits. *Linux J., 2009*(184), 5.

Blackwell, C. (2009). *A security architecture to protect against the insider threat from damage, fraud and theft*. Paper presented at the Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, Oak Ridge, Tennessee.

Boström, G. (2004). *Database Encryption as an Aspect*. Paper presented at the Workshop on AOSD Technology for Application-level Security, United Kingdon.

Boyen, X. (2009). *Hidden Credential Retrieval from a Reusable Password*. Paper presented at the Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, Sydney, Australia.

Brandel, M. (2009). Source Code Analysis Tools: How to Choose and Use Them. *CSO*. Retrieved from http://www.csoonline.com/article/print/477016

Bresz, F., Renshaw, T., Rozek, J., & White, T. (2007). Global Technology Audit Guide *Identity and Access Management* Retrieved from http://www.theiia.org/guidance/standards-and-guidance/ippf/practice-guides/gtag/gtag9/

Byres, E. J., Franz, M., & Miller, D. (2004). *The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems*. Paper presented at the International Infrastructure Survivability Workshop (IISW), Lisbon, Portugal.

CAPEC-1000: Mechanism of Attack. (2011). Retrieved January 15, 2012, 2012, from http://capec.mitre.org/data/graphs/1000.html

Cappelli, D., Moore, A., Shimeall, T. J., & Trzeciak, R. (2009). *Common Sense Guide to Prevention and Detection of Insider Threats*. Carnegie Mellon University Retrieved from http://www.cert.org/archive/pdf/CSG-V3.pdf.

Chen, B., Curtmola, R., Ateniese, G., & Burns, R. (2010). *Remote data checking for network coding-based distributed storage systems*. Paper presented at the Proceedings of the 2010 ACM workshop on Cloud computing security workshop, Chicago, Illinois, USA.

Chhabra, S., Rogers, B., Solihin, Y., & Prvulovic, M. (2011). *SecureME: a hardware-software approach to full system security*. Paper presented at the Proceedings of the international conference on Supercomputing, Tucson, Arizona, USA.

Chhabra, S., & Solihin, Y. (2011). i-NVMM: a secure non-volatile main memory system with incremental encryption. *SIGARCH Comput. Archit. News, 39*(3), 177-188. doi: 10.1145/2024723.2000086

Chinchani, R., Iyer, A., Ngo, H. Q., & Upadhyaya, S. (2005). *Towards a Theory of Insider Threat Assessment*.

Chumash, T., & Yao, D. (2009). Detection and Prevention of Insider Threats in Database Driven Web Services. In E. Ferrari, N. Li, E. Bertino & Y. Karabulut (Eds.), *Trust Management III* (Vol. 300, pp. 117-132): Springer Boston.

Cloakware Password Authority™. (2009). Retrieved from http://datacenter.cloakware.com/support/resources.php#whitepapers

Collberg, C. (2011). Toward Digital Asset Protection*, 26,* 8-13.

Dowd, M., McDonald, J., & Schuh, J. (2007). *The Art of Software Security Assessment*. Boston, MA: Addison-Wesley.

Edge, C., & Mitropoulos, F. (2009). *Aspectization of the Secure Communication Pattern for Data Integrity*. Paper presented at the Association of Information Systems SIGSEC Workshop on Information Security & Privacy (WISP 2009), Phoenix, AZ.

Enck, W., Butler, K., Richardson, T., Patrick, M., & Adam, S. (2008). *Defending Against Attacks on Main Memory Persistence*.

Encryption Technology for HP StorageWorks LTO Ultrium Tape Drives. (2010). Hewlett-Packard Development Company, L.P.

Englert, B., & Shah, P. (2009). *On the design and implementation of a secure online password vault*. Paper presented at the Proceedings of the 2009 International Conference on Hybrid Information Technology, Daejeon, Korea.

Falcarin, P., Collberg, C., Atallah, M., & Jabubowski, M. (2011). Guest Editors' Introduction: Software Protection*, 28,* 24-27.

Fendler, P. (2004). *Securing varieties of file systems*. Paper presented at the Proceedings of the 1st annual conference on Information security curriculum development, Kennesaw, Georgia.

Filman, R. E., Elrad, T., Clarke, S., & Aksit, M. (2005). *Aspect-Oriented Software Development*: Addison-Wesley.

Franqueira, V. N. L., Cleeff, A. v., Eck, P. v., & Wieringa, R. (2010). *External Insider Threat: A Real Security Challenge in Enterprise Value Webs*.

Futcher, L., & Solms, R. v. (2008). *Guidelines for secure software development*. Paper presented at the Proceedings of the 2008 annual research conference of the South

African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, Wilderness, South Africa.

Ge, T., & Zdonik, S. (2007). *Answering aggregation queries in a secure system model*. Paper presented at the Proceedings of the 33rd international conference on Very large data bases, Vienna, Austria.

George, B., & Valeva, A. (2006). *A database security course on a shoestring*. Paper presented at the Proceedings of the 37th SIGCSE technical symposium on Computer science education, Houston, Texas, USA.

Giacobazzi, R., Jones, N. D., & Mastroeni, I. (2012). *Obfuscation by partial evaluation of distorted interpreters*. Paper presented at the Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, Philadelphia, Pennsylvania, USA.

Gligor, V. D. (1998). *On the Formal Definition of Separation-of-Duty Policies and their Composition*.

Guimaraes, M., Murray, M., & Austin, R. (2007). *Incorporating database security courseware into a database security class*. Paper presented at the Proceedings of the 4th annual conference on Information security curriculum development, Kennesaw, Georgia.

Hansman, S., & Hunt, R. (2005). A taxonomy of network and computer attacks. *Computers & Security, 24*(1), 31-43. doi: 10.1016/j.cose.2004.06.011

Hargreaves, C., & Chivers, H. (2008). *Recovery of Encryption Keys from Memory Using a Linear Scan*.

Hicks, B., Rueda, S., St.Clair, L., Jaeger, T., & McDaniel, P. (2007). *A logical specification and analysis for SELinux MLS policy*. Paper presented at the Proceedings of the 12th ACM symposium on Access control models and technologies, Sophia Antipolis, France.

Hongyu, G., Hu, J., Huang, T., Wang, J., & Chen, Y. (2011). Security Issues in Online Social Networks*, 15,* 56-63.

Howard, J. D., & Longstaff, T. A. (1998). A Common Language for Computer Security Incidents: Sandia National Laboratories.

Insider threat study: illicit cyber activity in the banking and finance sector. (2004). from http://www.cert.org/archive/pdf/bankfin040820.pdf

ISACA. (2007). COBIT (Objectives for Information and related Technology) (Vol. AI3.4, AI7.4): IT Governance Institute.

ISO. (2005). ISO/IEC 27001:2005 *Information technology -- security techniques -- information security management systems -- requirements*: International Organization for Standards (ISO).

Jaeger, T., & Tidswell, J. E. (2001). Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur., 4*(2), 158-190. doi: http://doi.acm.org/10.1145/501963.501966

Jerbi, A., Hadar, E., Gates, C., & Grebenev, D. (2008). *An access control reference architecture*. Paper presented at the Proceedings of the 2nd ACM workshop on Computer security architectures, Alexandria, Virginia, USA.

Jianfeng, L. (2009). *Dynamic Enforcement of Separation-of-Duty Policies*.

Johnson, H. J., Gu, Y., & Chow, S. T. (1999). CA Patent No. 2340742. C. I. P. Office.

Kepner, C. H., & Tregoe, B. B. (1981). The Uses of Decision Analysis *The New Rational Manager* (pp. 103-105). Princeton, NJ: Princeton Research Press.

Kher, V., & Kim, Y. (2005). *Securing distributed storage: challenges, techniques, and systems*. Paper presented at the Proceedings of the 2005 ACM workshop on Storage security and survivability, Fairfax, VA, USA.

Kostiainen, K., Ekberg, J.-E., Asokan, N., & Rantala, A. (2009). *On-board credentials with open provisioning*. Paper presented at the Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, Sydney, Australia.

Libes, D. (1990, June 1990). *expect: Curing Those Uncontrollable Fits of Interaction.* Paper presented at the 1990 USENIX Conference, Anaheim, CA.

Libes, D. (1993). Kibitz - connecting multiple interactive programs together. *Software - Practice and Experience, 23*(5), 465-475.

Libes, D. (1994a). *Handling Passwords with Security and Reliability in Background Processes.* Paper presented at the Eighth Systems Administration Conference, San Diego, CA.

Libes, D. (1994b). *X Wrappers for Non-Graphic Interactive Programs.* Paper presented at the Xhibition 94,, San Diego, CA.

Lieberman, G. (2010). A False Sense of Security *Internal Auditor Online* (June 12, 2010 ed.): The Institute of Internal Auditors. Inc.

Majumdar, A., Drape, S. J., & Thomborson, C. D. (2007). *Slicing obfuscations: design, correctness, and evaluation*. Paper presented at the Proceedings of the 2007 ACM workshop on Digital Rights Management, Alexandria, Virginia, USA.

Making Security a Business Priority. (2008). *ComputerWorld Hong Kong, 25*(6), 7-9.

Managing Embedded Application Passwords with Password Auto Repository™ (PAR). (2009). Retrieved from http://www.e-dmzsecurity.com/pdf/e-DMZ_PAR-AppPasswordMgt_WP.pdf

Martin, A. (2008). Define Segregation of Duties. *USBanker, 118*(12), 1.

Martin, B., Brown, M., & Paller, A. (2009). 2009 CEW/SANS top 25 most dangerous programming errors. *Common Weakness Enumeration (CWE)*, from http://cwe.mitre.org/top25/index.html

Mattsson, U. T. (2008). How to Prevent Internal and External Attacks on Data - Securing the Enterprise Data Flow Against Advanced Attacks. *SSRN eLibrary*.

Mavrikidis, J. J. (1996). Security issues in a networked UNIX and MVS/VM environment. *SIGSAC Rev., 14*(3), 2-8. doi: http://doi.acm.org/10.1145/236397.236399

Muller, T., Freiling, F. C., & Dewald, A. (2011). *TRESOR runs encryption securely outside RAM*. Paper presented at the Proceedings of the 20th USENIX conference on Security, San Francisco, CA.

Nilsen, K. (2010). Keeping Fraud in the Cross Hairs. *Journal of Accountancy, 209*(6), 6.

OWASP. (2007). The ten most critical web application security vulnerabilities: Open Web Application Security Project (OWASP).

Page, A. E., & Marinov, S. (2007). Retrieved January 22/2012, 2012, from http://ptkdb.sourceforge.net/about.html

Password Management API for Application-to-Application Password Management. (2009). Retrieved from http://www.manageengine.com/products/passwordmanagerpro/help/index.html

Payne, C. (2007). *A cryptographic access control architecture secure against privileged attackers*. Paper presented at the Proceedings of the 2007 ACM workshop on Computer security architecture, Fairfax, Virginia, USA.

PCI. (2009). Payment Card Industry (PCI) Data Security Standard *Requirements and Security Assessment Procedures* (pp. 74): The Payment Card Industry Security Standards Council.

Randazzo, M. R., Keeney, M., Kowalski, E., Cappelli, D., & Moore, A. (2005). Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector (pp. 37).

Rechtman, Y. (2009). Evaluating Software Risk as Part of a Financial Audit. *The CPA Journal, 79*(6), 4.

Sade, Y., & Adar, R. (2008). US Patent No. 20080196101.

Said, H. E., Guimaraes, M. A., Maamar, Z., & Jololian, L. (2009). *Database and database application security*. Paper presented at the Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, Paris, France.

Salem, M. B., Hershkop, S., & Stolfo, S. J. (2008). A Survey of Insider Attack Detection Research *Insider Attack and Cyber Security* (Vol. 39, pp. 69-90).

Salerno, S. (2009). Criminal Injustice. *Skeptic, 15*(1), 8.

Sarbanes-Oxley section 404: A Guide for Management by Internal Controls Practitioners. (2008). Retrieved from http://www.theiia.org/download.cfm?file=31866

Sheppard, D. (2000). Beginner's Introduction to Perl Retrieved March 10, 2012, 2012, from http://www.perl.com/pub/2000/10/begperl1.html

Shiflett, C. (2004). Shared Hosting. *php|architect, III*.

Shmueli, E., Vaisenberg, R., Elovici, Y., & Glezer, C. (2010). Database encryption: an overview of contemporary challenges and design considerations. *SIGMOD Rec., 38*(3), 29-34. doi: 10.1145/1815933.1815940

Singleton, T. (2002). Stop fraud cold with powerful internal controls. *The Journal of Corporate Accounting & Finance, 13*(4), 29-40.

Sosonkin, M., Naumovich, G., & Memon, N. (2003). *Obfuscation of design intent in object-oriented applications*. Paper presented at the Proceedings of the 3rd ACM workshop on Digital rights management, Washington, DC, USA.

Weber, S., Karger, P. A., & Paradkar, A. (2005). A software flaw taxonomy: aiming tools at security. *SIGSOFT Softw. Eng. Notes, 30*(4), 1-7. doi: 10.1145/1082983.1083209

Woodbury, C. (2005). Eight Reasons to Stop Ignoring the Security of Your Development Systems. *IBM Systems magazine*.

Yang, L. (2009). *Teaching database security and auditing*. Paper presented at the Proceedings of the 40th ACM technical symposium on Computer science education, Chattanooga, TN, USA.

Yu, T., Sivasubramanian, D., & Xie, T. (2009). *Security policy testing via automated program code generation*. Paper presented at the Proceedings of the 5th Annual

Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, Oak Ridge, Tennessee.

Yu, T., Winslett, M., & Seamons, K. E. (2003). Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies for Automated Trust Negotiation. *ACM Trans. Inf. Syst. Secur., 6*(1), 1-42. doi: 10.1145/605434.605435

Zaobin, G., Tang, J., Wu, P., & Varadharajan. (2007). *A Novel Security Risk Evaluation for Information Systems*.

Zhu, X., Feng, H., & Chen, H. (2009). *Access Control Policy Based on Behavior Patterns*.