Countering Network Worms Through Automatic Patch Generation*

Stelios Sidiroglou
Columbia University
stelios@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

We propose the first end-point architecture for automatically repairing software flaws such as buffer overflows that are exploited by zero-day worms. Our approach relies on source code transformations to quickly apply automatically-created (and tested) patches to vulnerable segments of the targeted applications, exploiting the fact that a worm must reveal its infection vector to achieve further infection. Preliminary experimental results indicate a success rate of 82%, and a repair time of 3 seconds.

Keywords: Worms, viruses, honeypots, software patching, process sandboxing, memory protection, buffer overflows.

1 Introduction

Recent incidents have demonstrated the ability of self-propagating code, also known as "network worms," to infect large numbers of hosts, exploiting vulnerabilities in the largely homogeneous deployed software base [1] [16, 8], often affecting the off-line world in the process [5]. Even when a worm carries no malicious payload, the direct cost of recovering from the side effects of an infection epidemic can be tremendous. Thus, countering worms has recently become the focus of increased research, generally focusing on content-filtering mechanisms combined with large-scale coordination strategies [9, 13].

Despite some promising early results, we believe that this approach will be insufficient by itself in the future. We base this primarily on two observations. First, to achieve coverage, such mechanisms are intended for use by routers (*e.g.,* Cisco's NBAR); given the routers' limited budget in terms of processing cycles per packet, even mildly polymorphic worms (mirroring the evolution of polymorphic viruses, more than a decade ago) are likely to evade such filtering, as demonstrated recently in [2]. Network-based intrusion detection systems (NIDS) have encountered similar problems, requiring fairly invasive packet processing and queuing at the router or firewall. When placed in the application's critical path, as such filtering mechanisms must, they will have an adverse impact on performance. Second, end-to-end "opportunistic"[2] encryption in the form of TLS/SSL or IPsec is being used by an increasing number of hosts and applications. We believe that it is only a matter of time until worms start using such encrypted channels to cover their tracks. *Encryption introduces a much greater problem than filtering since it is inherently insolvable.* Similar to the case for distributed firewalls, these trends argue for an end-point worm-countering mechanism.

A preventative approach to the worm problem is the elimination or minimization of remotely-exploitable vulnerabilities, such as buffer overflows. Detecting potential buffer overflows is a very difficult problem, for which only partial solutions exist. "Blanket" solutions such as StackGuard or MemGuard [3] typically exhibit at least one of two problems: reduced system performance, and overflows are detected after they have occurred and have overflown information in the stack so continuation is not possible. Thus, they are inappropriate for high-performance, high-availability environments such as a heavily-used e-commerce web server. Similarly, type-safe retrofitting of code and

*An expanded version can be found as Columbia University Computer Science Technical Report CUCS-029-03.

[1] `http://www.siliconedefense.com/research/worms/slammer`

[2] By "opportunistic" we mean that client-side, and often server-side, authentication is often not strictly required, as is the case with the majority of web servers or with SMTP over TLS (*e.g.,* sendmail's STARTTLS option).

fault isolation techniques incur similar performance degradation and in some cases are inapplicable. An ideal solution would use expensive protection mechanisms only where needed and allow applications to gracefully recover from such attacks. La Brea[3] attempts to slow the growth of TCP-based worms by accepting connections and then blocking on them indefinitely, causing the corresponding worm thread to block. Unfortunately, worms can avoid these mechanisms by probing and infecting asynchronously. Under the connection-throttling approach [14], each host restricts the rate at which connections may be initiated. If adopted universally, such an approach would reduce the spreading rate of a worm by up to an order of magnitude, without affecting legitimate communications.

Another widespread protection mechanism to guard against worms has traditionally been the use of anti-virus software. The obvious problem with this approach is that it primarily depends on up-to-date virus definitions which in turn rely, for the most part, on human intervention. The non real-time characteristics of this approach render it highly ineffective against zero-day worms (or "effectively zero-day" worms, such as Witty, which appeared 1 day after the exploited vulnerability was announced). Likewise, patch management is ineffective against zero-day worms; even as a tool against well known vulnerabilities, it has been shown to be little-used in practice [11].

We propose an end-point first-reaction mechanism that tries to automatically patch vulnerable software by identifying and transforming the code surrounding the exploited software flaw. Our approach focuses on buffer overflows, as they constitute the vast majority of non-email worm infection vectors. Briefly, we use instrumented versions of an enterprise's important services (*e.g.*, web server) in a sandboxed environment. This environment is operated *in parallel with* the production servers, and is not used to serve actual requests. Instead, we use it as a "clean room" environment to test the effects of "suspicious" requests, such as potential worm infection vectors in an asynchronous fashion (*i.e.*, without necessarily blocking requests). A request that causes a buffer overflow on the production server will have the same effect on the sandboxed version of the application. Appropriate instrumentation allows us to determine the buffers and functions involved in a buffer overflow attack. We then apply several source-code transformation heuristics that aim to contain the buffer overflow. Using the same sandboxed environment, we test the patches against both the infection vectors and a site-specific functionality test-suite, to identify any obvious problems. It is also possible to add a comprehensive test suite that would possibly take tens of minutes to complete, at the expense of quick reaction (although we can block access to the vulnerable service with a firewall rule, while the tests are underway). Even so, our approach would be a few orders of magnitude faster than waiting for a "proper" fix. If the tests are successful, we restart the production servers with the new version of the targeted program. We are careful to produce localized patches, for which we can be fairly confident that they will not introduce additional instabilities, although we (obviously) cannot offer any guarantees; additional research is needed in this area. Note that the patch generation and testing occurs in a completely *decentralized* and *real-time* fashion, without need for a centralized update site, which may become an attack target, as happened with the W32/Blaster worm.

Our architecture makes use of several components that have been developed for other purposes. Its novelty lies in the combination of all the components in fixing vulnerable applications without unduly impacting their performance or availability. Our major assumption is that we can extract a worm's infection vector (or, more generally, one instance of it, for polymorphic worms). As we discuss in Section 2, we envision the use of various mechanisms such as honeypots, host-based, and network-based intrusion detection sensors. Note that vector-extraction is a necessary precondition to any reactive or filtering-based solution to the worm problem. An example of an appropriate mechanism for vector-extraction is the Anomalous Payload-based Network Intrusion Detection system [15].

A secondary assumption is that the source code for the application is available. Although our architecture can use binary rewriting techniques, in this paper we focus on source-code transformations. We should also note that several popular server applications are in fact open-source (*e.g.*, Apache, Sendmail, MySQL, Bind). Furthermore, although our architecture can be used verbatim to react to lower-intensity "hack-in" attempts, in this paper we focus on the higher-profile (and easier to detect) worm problem.

To determine the effectiveness of our approach, we tested a set of 17 applications vulnerable to buffer overflows, compiled by the Cosak project[4]. We simulated the presence of a worm (even for those applications that were not in fact network services) by triggering the buffer overflow that the worm would exploit to infect the process. Our proof-of-concept evaluation experiments show that our architecture was able to fix the problem in 82% of all cases. An experiment with a hypothetical vulnerability in the Apache web server showed that the total time to produce a

[3] `http://www.threenorth.com/LaBrea/LaBrea.txt`
[4] `http://serg.cs.drexel.edu/cosak/index.shtml`

correct and tested fix was 3 seconds. This means that the total cycle from detection to updating the real server can be less than 10 seconds. Although there is no guarantee that the automatically generated patches cannot themselves cause damage, in practice they were sufficient to contain these attacks without harmful side-effects. When concerns about safety override security, it is possible to add a human in the loop (*e.g.*, by popping up a window displaying the vulnerability and the suggested fix, asking for approval); although this would slow down the reaction time, it would still be a few orders of magnitude faster than waiting for an official patch.

What must be stressed here is that although it takes a few seconds to create and test a patch, throttling a worm's infection rate can begin immediately after a vulnerability has been detected by either halting the production server while the vulnerability is being patched or by setting a firewall rule that drops all traffic to and from the production server. When throttling and patching are incorporated into the *logistic* equation [13] one can clearly see that our approach helps decelerate the initial spread of a worm.

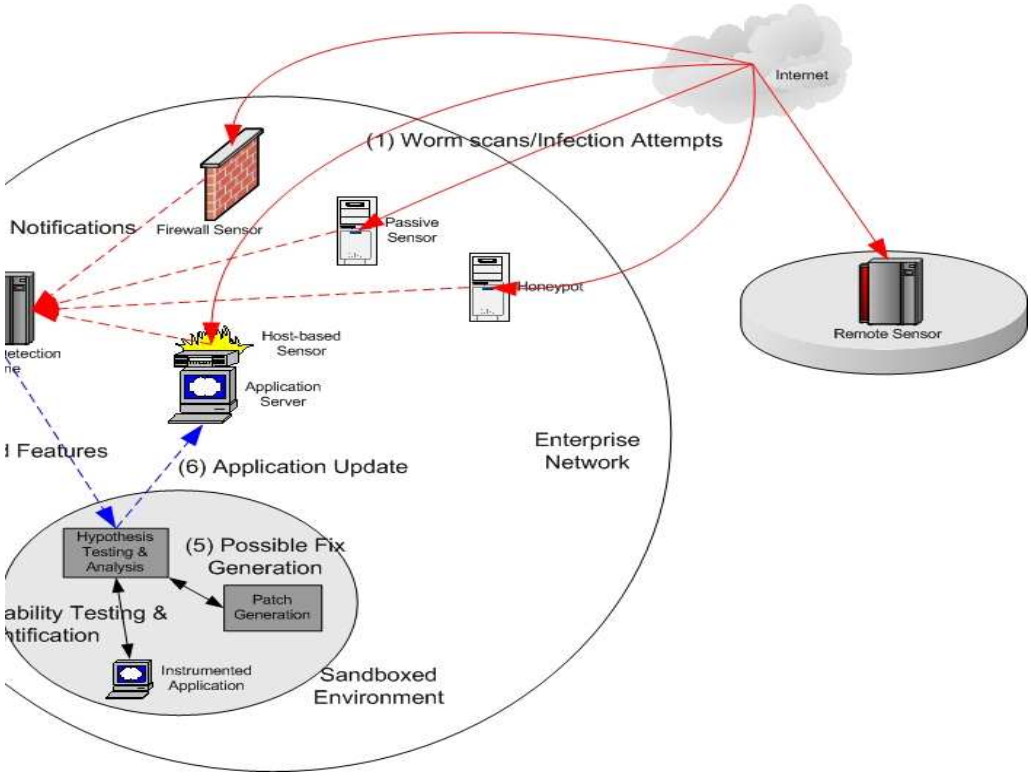

Figure 1. **Worm vaccination architecture: sensors deployed at various locations in the network detect a potential worm (1), notify an analysis engine (2) which forwards the infection vector and relevant information to a protected environment (3). The potential infection vector is tested against an appropriately-instrumented version of the targeted application, identifying the vulnerability (4). Several software patches are generated and tested using several different heuristics (5). If one of them is not susceptible to the infection and does not impact functionality, the main application server is updated (6).**

2 System Architecture

Our architecture, depicted in Figure 1, makes use of five types of components: a set of worm-detection sensors, a correlation engine, a sandboxed environment running appropriately-instrumented versions of the applications used in the enterprise network (*e.g.*, Apache web server, SQL database server, *etc.*), an analysis and patch-generation engine, and a software update component. We now describe each of these components.

2.1 Worm Detection and Correlation Engine

The worm-detection sensors are responsible for detecting potential worm probes and, more importantly, infection attempts. Several types of sensors may be employed concurrently:

- Host-based sensors, monitoring the behavior of deployed applications and servers.
- Passive sensors on the corporate firewall or on independent boxes, eavesdropping on traffic from/to the servers.
- Honeypots that simulate the behavior of the target application and capture any communication.
- Other types of sensors, including sensors run by other entities (more on this in Section 5).

Any combination of the above sensors can be used simultaneously, although we believe honeypot servers are the most promising, since worms cannot distinguish between real and fake servers in their probes. Honeypots and other deception-based mechanisms are also effective against hit-list-based worm propagation strategies, assuming they have been in place during the scanning phase [13].

These sensors communicate with each other and with a central server, which correlates events from independent sensors and determines potential infection vectors (*e.g.*, the data on an HTTP request, as seen by a honeypot). In general, we assume that a worm can *somehow* be detected. We believe our assumption to be pragmatic, in that most current solutions to the worm problem depend on a satisfactory answer to the problem of worm detection. A possible mechanism that can be employed for this purpose is the payload-based anomaly detector (PAYL) [15].

PAYL models the normal application payload of network traffic in a fully automated, unsupervised fashion. PAYL uses the Mahalanobis distance during the detection phase to calculate the similarity of new data against the precomputed profile. The advantages of this method are multi-fold; it is state-less, it does not parse the input stream, it generates a small model which can be updated using an incremental online learning algorithm maintaining an accurate model in real-time. We stress the importance of real-time mechanisms such as PAYL since it provides a mechanism with which we can contain Warhol worms (hit-list worms).

The purpose of the correlation engine is to determine the likelihood of any particular communication being an infection vector (or a manually-launched attack) and to request that the suspicious communication be tested in the sandbox environment. It also maintains a list of fixed exploits and of false positives (communications that have already been determined to be innocuous, or at least ineffective against the targeted application).

2.2 Sandboxed Environment

The potential infection vector (*i.e.*, the byte stream which, when "fed" to the target application, will cause an instance of the worm to appear on the target system) is forwarded to a sandboxed environment, which runs appropriately-instrumented instances of the applications we are interested in protecting (*e.g.*, Apache or IIS server). The instrumentation can be fairly invasive in terms of performance, since we only use the server for clean-room testing. The performance degradation from using a mechanism like ProPolice may introduce an overhead of as much as an order of magnitude which, although prohibitive in productions systems, is acceptable for testing purposes. In its most powerful form, a full-blown machine emulator can be used to determine whether the application has been subverted. Other potential instrumentation includes light-weight virtual machines, dynamic analysis/sandboxing tools, or mechanisms such as MemGuard. These mechanisms are generally not used for application protection due to their considerable impact on performance and the fact that they typically cause the application to "fault", rather than continue operation. In our approach, this drawback is not of particular importance because we only use these mechanisms in the sandboxed environment to identify as accurately as possible the source of weakness in the application. These mechanisms are not used for the production servers. For example, MemGuard [3] can identify both the specific buffer and function that is exploited in a buffer-overflow attack. Alternatively, when running under a simulator, we can detect when execution has shifted to the stack, heap, or some other unexpected location, such as an unused library function.

The more invasive the instrumentation, the higher the likelihood of detecting subversion and identifying the source of the vulnerability. Although the analysis step can only identify known classes of attack (*e.g.*, a stack-based buffer overflow), even if it can detect anomalous behavior, new classes of attack (*e.g.*, heap-based overflows) appear less often than exploits of known attack types. *Note that we are not limited to known exploits and attacks.*

2.3 Patch Generation and Testing

Armed with knowledge of the vulnerability, we can automatically patch the program. Generally, program analysis is an impossible problem (reducible to the *Halting* problem). However, there are a few fixes that may mitigate the effects of an attack:

- Moving the offending buffer to the heap, by dynamically allocating the buffer upon entering the function and freeing it at all exit points. Furthermore, we can increase the size of the allocated buffer to be larger than the size of the infection vector, thus protecting the application from even crashing (for fixed-size exploits). Finally, we can use a version of *malloc()* that allocates two additional write-protected pages that bracket the target buffer. Any buffer overflow or underflow will cause the process to receive a Segmentation Violation (SEGV) signal. This signal is caught by a signal handler we have added to the source code. The signal handler can then *longjmp()* to the code immediately after the routine that caused the buffer overflow. Although this approach could be used in a "blanket" manner (*i.e.*, applied everywhere in the code where a buffer overflow *could* occur, the performance implications would be significant. Instead, we use the worm's infection vector as a hint to locate the potential vulnerability, somewhat simplifying the problem. We give more details in Section 3.

- Add code that recognizes either the attack itself or specific conditions in the stack trace (*e.g.*, a specific sequence of stack records), and returns from the function if it detects these conditions. The former is in some sense equivalent to content filtering, and least likely to work against even mildly polymorphic worms. Generally, this approach appears to be the least promising.

- More generally, we can use some minor code-randomization techniques[1] that could "move" the vulnerability such that the infection vector no longer works.

- Finally, we can attempt to "slice-off" some functionality, by immediately returning from mostly-unused code that contains the vulnerability. Especially for large software systems that contain numerous, often untested, features that are not regularly used, this may be the solution with the least impact. We can determine whether a piece of functionality is unused by profiling the real application; if the vulnerability is in an unused section of the application, we can logically remove that part of the functionality (*e.g.*, by an early function-return).

We focus on the first approach, as it seems the most promising. We plan to further investigate other heuristics in future research. The patches we introduce are localized, to avoid introducing additional instability to the application. Although it is very difficult, if not impossible, to argue about the correctness of any newly introduced code (whether it was created by a human or an automated process such as ours), we are confident that our patches do not exacerbate the problem because of their minimal scope and the fact that they emulate behavior that could have been introduced automatically by the compiler or some other automated tool during the code authoring or compilation phase. Although this is by no means a proof of correctness, we believe it is a good argument with respect to the safety of the approach.

Our architecture makes it possible to add new analysis techniques and patch-generation components easily. To generate the patches, we employ TXL [7], a language-independent code-transformation tool. We describe its use in more detail in Section 3.

We can test several patches (potentially in parallel) until we are satisfied that the application is no longer vulnerable to the specific exploit. To ensure that the patched version will continue to function, a site-specific test suite is used to determine what functionality (if any) has been lost. The test suite is generated by the administrator in advance, and should reflect a typical workload of the application, exercising all critical aspects (*e.g.*, performing purchasing transactions). Naturally, one possibility is that no heuristic will work, in which case it is not possible to automatically fix the application and other measures have to be used.

2.4 Application Update

Once we have a worm-resistant version of the application, we must instantiate it on the server. Thus, the last component of our architecture is a server-based monitor. To achieve this, we can either use a virtual-machine approach or assume that the target application is somehow sandboxed and implement the monitor as a regular process residing outside that sandbox. The monitor receives the new version of the application, terminates the running instance (first attempting a graceful termination), replaces the executable with the new version, and restarts the server.

3 Implementation

Our prototype implementation is comprised of three components: ProPolice, TXL, and a sandboxed environment. These components interact to identify software vulnerabilities, apply potential patches, and provide a secure environment respectively. In Section 4 we use the implementation to simulate attacks and provide fixes for a sample service application and a list of vulnerable open-source products compiled by the Code Security Analysis Kit (CoSAK) project. Here, we introduce the components and discuss the implementation.

3.1 ProPolice

In order to detect the source of buffer overflow/underflow vulnerabilities, we employ the OpenBSD version of ProPolice [5], a GCC extension for protecting applications from stack-smashing attacks in a manner similar to StackGuard [3]. ProPolice will return the names of the function and offending buffer that lead to the overflow condition. This information is then forwarded to a TXL program that attempts a number of heuristics, as discussed in Section 2.

ProPolice is a GCC extension for protecting applications from stack-smashing attacks. The protection is realized by buffer overflow detection and the variable reordering feature to avoid the corruption of pointers. Its novel features are (1) the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations, (2) the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to corrupt arbitrary memory locations further, and (3) the omission of instrumentation code from some functions to decrease the performance overhead.

When a buffer overflow attack is attempted on applications compiled with the ProPolice extensions, the execution of the program is interrupted and the offending function and buffer are reported. When used to protect a service, ProPolice incurs a modest performance overhead, similar to StackGuard's [3]. More importantly, the application under attack detects overflows after they have occurred and have subsequently overflown information in the stack, making continuation of the program undesirable. While this is more palatable than outright subversion, it is sub-optimal in terms of service availability.

Better mechanisms to use include Valgrind[6]or MemGuard [3]. Although ProPolice was sufficient for our prototype implementation, a fully-functional system would use either of these systems to catch *all* illegal memory-dereferences (even those in the heap). Both of these systems are considerably slower than ProPolice, capable of slowing down an application by even an order of magnitude, making them unsuitable for use by production systems but usable in our system. Fortunately, their impact on performance is less relevant in our approach.

3.2 TXL

Armed with the information produced by ProPolice, the code-transformation component of our system, TXL, is invoked. TXL is a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Nauer (BNF). Several parsing strategies are supported by TXL making it comfortable with ambiguous grammars allowing for more "natural" user-oriented grammars, circumventing the need for strict compiler-style "implementation" grammars [7].

In our system, we use TXL for C-to-C transformations by making changes to the ANSI C grammar. In particular we move to the heap variables originally defined on the stack, using a simple TXL program. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the C grammar to free the variables before the function returns. After making changes to the standard ANSI C grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial. The other heuristic we use is "slice-off" functionality. There, we use TXL to simply comment out the code of the superfluous function and embed a "return" in the function.

In the move-to-heap approach, we use an alternative *malloc()* implementation we developed specifically for this purpose. *pmalloc()* allocates two additional, zero-filled write-protected memory pages that surround the requested allocated memory region, as shown in Figure 2. Any buffer overflow or underflow will cause the operating system to

[5] `http://www.trl.ibm.com/projectes/security/spp/`
[6] `http://valgrind.kde.org`

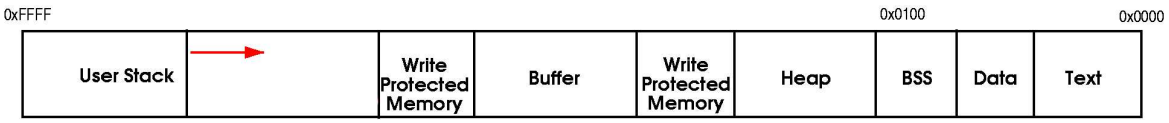

Figure 2. **Protected Malloc: Write-protected memory pages surround a buffer allocated with** *pmalloc().*

issue a Segmentation Violation signal (SIGSEGV) to the process. We use *mprotect()* to mark the surrounding pages as read-only.

Our TXL program inserts a *setjmp()* call immediately before the function call that caused the buffer overflow. The effect of this operation is to save the stack pointers, registers, and program counter, such that the program can later restore their state. We also inject a signal handler that catches the SIGSEGV and calls *longjmp()*, restoring the stack pointers and registers (including the program counter) to their values prior the call to the offending function (in fact, they are restored to their values as of the call to *setjmp()*). The program will then re-evaluate the injected conditional statement that includes the *setjmp()* call. This time, however, the return value will cause the conditional to evaluate to false, thereby skipping execution of the offending function. Note that the targeted buffer will contain exactly the amount of data (infection vector) it would if the offending function performed correct data-truncation.

There are two benefits in this approach. First, objects in the heap are protected from being overwritten by an attack on the specified variable, since there is a signal violation when data is written beyond the allocated space. Second, we can recover gracefully from an overflow attempt, since we can recover the stack context environment prior to the offending function's call, and effectively *longjmp()* to the code immediately following the routine that caused the overflow or underflow.

Examination of the source code of the programs featured in the CoSAK study illustrated that the majority of the calls that caused an overflow/underflow (*e.g., strcpy(), memcpy(), etc.*) did not check for return values or include calls to other routines. This is an important observation since it validates our assumption that the heuristic can circumvent the malignant call using *longjmp()*.

3.3 Sandboxed Environment

Finally, for our sandboxed environment we use the VMWare virtual machine where we run the OpenBSD operating system. VMWare allows operating systems and software applications to be isolated from the underlying operating system in secure virtual machines that co-exist on a single piece of hardware. Once we have created a correct version of the application, we simply update its image on the production environment outside the virtual environment, and restart it.

4 Experimental Evaluation

In order to illustrate the capabilities of our system and the effectiveness of the patch heuristics, we constructed a simple file-serving application that had a buffer overflow vulnerability and contained superfluous services where we could test against stack-smashing attacks and slice-of functionality respectively. For these purposes, the application used a simple two-phase protocol where a service is requested (different functions) and then the application waits for network input. The application was written in ANSI C.

A buffer overflow attack was constructed that overwrites the return address and attempts to get access to a root shell. The application was compiled under OpenBSD with the ProPolice extensions to GCC. Armed with the knowledge provided by ProPolice, the names of the function and buffer potentially responsible for the buffer overflow, the TXL implementation of our heuristics is invoked. Specific to the set of actions that we have implemented thus far, we test the heuristics and recompile the TXL-transformed code, and run a simple functionality test on the application (whether it can correctly serve a given file). The test is a simple script that attempts to access the available service. This application was an initial proof-of-concept for our system, and did not prove the correctness of our approach. More substantial results were acquired through the examination of the applications provided by the Code Security Analysis Kit project.

4.1 CoSAK data

In order to further test our heuristics, we examined a number of vulnerable open-source software products. This data was made available through the Code Security Analysis Kit (CoSAK) project from the software engineering research group at Drexel university. CoSAK is a DARPA-funded project that is developing a toolkit for software auditors to assist with the development of high-assurance and secure software systems. They have compiled a database of thirty OSS products along with their known vulnerabilities and respective patches. This database is comprised of tools, products and libraries with known vulnerabilities, with a large number listed as susceptible to buffer overflow attacks.

We tested the move-to-heap heuristic against the CoSAK data-set, which resulted in fixing 14 out of 17 ”fixable” buffer overflow vulnerabilities, or 82% success rate. The remaining 13 products were not tested because their vulnerabilities were unrelated (non buffer-overflow). The products that were not amenable to the heuristic were examined, and in all cases what would be required to provide an appropriate fix would be adjustments to the TXL heuristics to cover special cases.

The majority of the vulnerabilities provided by the CoSAK dataset were caused by calls to the *strcpy()* routine. Examination of the respective security patches showed that for most cases the buffer overflow susceptibility could be repaired by a respective *strncpy()*. Furthermore, most routines did not check for return values and did not include routines within the routines, thus providing fertile ground for use of our *pmalloc()* heuristic.

4.2 Performance

In order to evaluate the performance of our system, we tested the patch-generation engine on an instrumented version of Apache 2.0.48. Apache was chosen due to its popularity and source-code availability. Basic Apache functionality was tested, omitting additional modules. The purpose of the evaluation was to validate the hypothesis that heuristics can be applied and tested in a time-efficient manner. The tests were conducted on a PC with an AMD Athlon processor operating at 2.8 GHz and 2 GB of RAM. The underlying operating system was OpenBSD 3.3.

One assumption that our system makes is that the instrumented application is already compiled in the sandboxed environment so that any patch heuristic would not require a complete re-compilation of the application. In order to get a realistic insight on the time that would be required from applying a patch and being able to test the application, we applied our move-to-heap TXL transformation on a number of different files, ranging from large to small sizes, and recompiled the latest version of Apache. The average over the different files for compilation and relinking was 2.5 seconds.

Another important issue in terms of performance is the TXL transformation time for our basic heuristics. By being able to pass the specific function name and buffer to TXL, the transformation time is greatly reduced as the rule-set is concentrated on a targeted section of the source code. The average transformation time for different functions that were examined was 0.045 seconds. This result is very encouraging as it allows the assumption that the majority of the heuristics can be applied and tested in under 3 seconds.

5 Discussion

5.1 Challenges

There are several challenges associated with our approach:

1. **Determination of the nature of the attack (*e.g.*, buffer overflow), and identification of the likely software flaws that permit the exploit.** Obviously, our approach can only fix already-known attacks, *e.g.*, stack or heap-based buffer overflows. This knowledge manifests itself through the debugging and instrumentation of the sandboxed version of the application. Currently, we use ProPolice[7] to identify the likely functions and buffers that lead to the overflow condition. More powerful analysis tools can be employed in our architecture to catch more sophisticated code-injection attacks, and we intend to investigate them in future work. One advantage of our approach is that the performance implications of such mechanisms are not relevant: an order of magnitude or more slow-down of the instrumented application is acceptable, since it does not impact the common-case usage. Furthermore, our architecture should be general enough that other classes of attack can be detected, *e.g.*, email worms, although we have not yet investigated this.

[7] `http://www.trl.ibm.com/projectes/security/spp/`

2. **Reliable repairing of the software.** Repairability is impossible to guarantee, as the general problem can be reduced to the Halting Problem. Our heuristics allow us to generate potential fixes for several classes of buffer overflows using code-to-code transformations [7], and test them in a clean-room environment. Further research is necessary in the direction of automated software recovery in order to develop better repair mechanisms. One interesting possibility is the use of Aspect-Oriented Programming [4] to create locations ("hooks") in the source code that would allow the insertion of appropriate fixes. We plan to investigate this in future research.

   Interestingly, our architecture could be used to automatically fix any type of software fault, such as invalid memory dereferences, by plugging-in the appropriate repair module. When it is impossible to automatically obtain a software fix, we can use content-filtering as in [12] to temporarily protect the service. The possibility of combining the two techniques is a topic of future research.

3. **Source-code availability.** Our system assumes that the source code of the instrumented application is available, so patches can be easily generated and tested. When that is not the case, binary-rewriting techniques [10] may be applicable, at considerably higher complexity. Instrumentation of the application also becomes correspondingly more difficult under some schemes. One intriguing possibility is that vendors ship two versions of their applications, a "regular" and an "instrumented" one; the latter would provide a standardized set of hooks that would allow a general monitoring module to exercise oversight.

4. Finally, with respect to multi-partite worms, *i.e.*, worms using multiple independent infection vectors and propagation mechanisms (*e.g.*, spreading over both email and HTTP), our architecture treats such infections as independent worms.

5.2 Centralized *vs.* Distributed Reaction

The authors of [13] envision a Cyber "Center for Disease Control" (CCDC) for identifying outbreaks, rapidly analyzing pathogens, fighting the infection, and proactively devising methods for detecting and resisting future attacks. However, it seems unlikely that there would ever be wide acceptance of an entity trusted to arbitrarily patch software running on any user's system. Furthermore, fixes would still need to be handcrafted by humans and thus arrive too late to help in worm containment. In our scheme, such a CCDC would play the role of a real-time alert-coordination and distribution system. Individual enterprises would be able to independently confirm the validity of a reported weakness and create their own fixes in a decentralized manner, thereby minimizing the trust they would have to place to the CCDC.

When an exploitable vulnerability is discovered, our architecture could be used by the CCDC to distribute "fake worms". This channel would be treated as another sensor supporting the analysis engine. Propagation of these fake worms would trigger the creation of a quick-fix if the warning is deemed authentic (*i.e.*, the application crashes as a result of running the attack in the sandbox). Again, this would serve as a mechanism for distributing quick patches by independent parties, by distributing only the exploit and allowing organizations to create their own patches. One proposal for such a channel, using a self-organizing overlay network is described in [6].

Note that although we speculate the deployment of such a system in every medium to large-size enterprise network, there is nothing to preclude pooling of resources across multiple, mutually trusted, organizations. In particular, a managed-security company could provide a quick-fix service to its clients, by using sensors in every client's location and generating patches in a centralized facility. The fixes would then be "pushed" to all clients. A similar approach is taken by some managed-security vendors, who keep a number of programmers available on a 24-hour basis. In all cases, administrators must be aware of the services offered (officially or unofficially) by all the hosts in their networks.

5.3 Attacks Against the System

Naturally, our system should not create new opportunities for attackers to subvert applications and hosts. One concern is the possibility of "gaming" by attackers, causing instability and unnecessary software updates. One interesting attack would be to cause oscillation between versions of the software that are alternatively vulnerable to different attacks. Although this may be theoretically possible, we cannot think of a suitable example. Such attack capabilities are limited by the fact that the system can test the patching results against both current and previous (but still pending, *i.e.*, not "officially" fixed by an administrator-applied patch) attacks. Furthermore, we assume that the various

system components are appropriately protected against subversion, *i.e.*, the clean-room environment is firewalled, the communication between the various components is integrity-protected using TLS/SSL or IPsec.

If a sensor is subverted and used to generate false alarms, event correlation will reveal the anomalous behavior. In any case, the sensor can at best only mount a denial of service attack against the patching mechanism, by causing many hypotheses to be tested. Again, such anomalous behavior is easy to detect and take into consideration without impacting either the protected services or the patching mechanism.

Another way to attack our architecture involves denying the communication between the correlation engine, the sensors, and the sandbox through a denial of service attack. Such an attack may in fact be a by-product of a worm's aggressive propagation, as was the case with the SQL worm [8]. Fortunately, it should be possible to ingress-filter the ports used for these communications, making it very difficult to mount such an attack from an external network.

As with any fully-automated task, the risks of relying on automated patching and testing as the only real-time verification techniques are not fully understood. To the extent that our system correctly determines that a buffer overflow attack is possible, the system's operation is safe: either a correct patch for the application will be created, or the application will have to be shut-down (or replaced with a non-working version). Considering the alternative, *i.e.*, guaranteed loss of service *and* subversion of the application, we believe that the risk will be acceptable to many. The question then centers around the correctness of the analysis engine. Fundamentally, this appears to be an impossible problem — our architecture enables us to add appropriate checks as needed, but we cannot guarantee absolute safety.

6 Conclusion

We argued that increased use of end-to-end encryption and worm stealthiness, as well as the inadequacy of existing preventive mechanisms to ensure service availability in the presence of software flaws, necessitate the development of an end-point worm-reaction approach that employs invasive but targeted mechanisms to fix the vulnerabilities. We presented an architecture for countering worms through automatic software-patch generation. Our architecture uses a set of sensors to detect potential infection vectors, and uses a clean-room (sandboxed) environment running appropriately-instrumented instances of the applications used in the enterprise network to test potential fixes. To generate the fixes, we use code-transformation tools to counter specific buffer-overflow instances. If we manage to create a version of the application that is both resistant to the worm and meets certain minimal-functionality criteria, embodied in a functionality test-suite created in advance by the system administrator, we update the production servers.

The benefits presented by our system are the quick reaction to attacks by the automated creation of 'good enough' fixes without any sort of dependence on a central authority, such as a hypothetical Cyber-CDC [13]. Comprehensive security measures can be administered at a later time. Furthermore, our architecture is easily extensible to accommodate detection and reactive measures against new types of attacks as they become known. Our experimental analysis, using a number of known vulnerable applications are hypothetical targets of a worm infection, shows that our architecture can fix 82% of all such attacks, and that the maximum time to repair a complicated application was less than 3 seconds. We believe that these preliminary results validate our approach and will spur further research.

Acknowledgements

We wish to thank the anonymous reviewers for their constructive comments and guidance during the preparation of this manuscript.

References

[1] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[2] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, August 2003.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[8] `http://www.siliconedefense.com/research/worms/slammer`

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[5] E. Levy. Crossover: Online Pests Plaguing the Offline World. *IEEE Security & Privacy*, 1(6):71–73, November/December 2003.

[6] J. Li, P. L. Reiher, and G. J. Popek. Resilient Self-Organizing Overlay Networks for Security Update Delivery. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1):189–202, January 2004.

[7] A. J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. *Computer Languages*, 19(3):157–168, 1993.

[8] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the 2nd Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[9] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[10] M. Prasad and T. Chiueh. A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.

[11] E. Rescorla. Security Holes...Who Cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 79–90, August 2003.

[12] J. C. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The Design and Implementation of an Intrusion Tolerant System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[13] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[14] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th USENIX Security Symposium*, pages 285–294, August 2003.

[15] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advanced in Intrusion Detection (RAID)*, pages 201–222, September 2004.

[16] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.