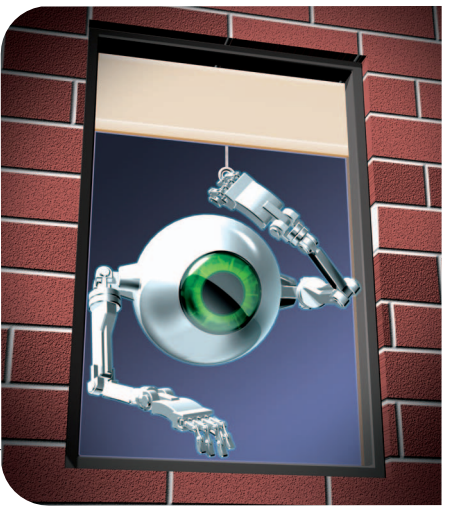
Countering Network Worms Through Automatic Patch Generation

To counter zero-day worms that exploit software flaws such as buffer overflows, this end-point architecture uses source code transformations to automatically create and test software patches for vulnerable segments of targeted applications.



Stelios Sidiroglou and Angelos D. Keromytis
Columbia University

Recent incidents have demonstrated the ability of self-propagating code, or *network worms*, to infect numerous hosts, exploiting vulnerabilities in the largely homogeneous deployed software base.[1,2] Such activities often have an impact on the offline world as well,[3] by indirectly affecting critical infrastructure such as the transportation, finance, and energy sectors. Even when a worm carries no malicious payload, there are tremendous direct recovery costs from an infection epidemic's side effects. Thus, researchers are increasingly focused on countering worms, typically using content-filtering mechanisms combined with large-scale coordination strategies.[4,5]

Given the routers' limited budget in terms of processing cycles per packet, however, even mildly polymorphic worms (mirroring the evolution of polymorphic viruses more than a decade ago) are likely to evade such filtering. Furthermore, hosts and applications are increasingly using end-to-end opportunistic encryption such as Transport Layer Security (TLS) and Secure Sockets Layer (SSL) or Internet Protocol security (IPsec). We believe that it's only a matter of time until worms start using such encrypted channels to cover their tracks. As with the case for distributed firewalls, these trends argue for an end-point worm-countering mechanism. By using invasive but targeted mechanisms to fix vulnerabilities, an end-point worm-reaction approach can use its strategic position to address vulnerabilities at their source, thus avoiding masking techniques that attackers can exploit to bypass detection.

To this end, we've developed an architecture that counters worms through automatic software-patch generation. Our approach uses sensors to detect potential infection vectors; to test potential fixes, we use a clean-room (sandboxed) environment running appropriately instrumented versions of network applications. The architecture generates fixes using code-transformation tools to counter specific buffer-overflow instances. If we can create an application version that's both worm resistant and meets some functionality criteria based on running predefined test suites, we update the production servers.

In proof-of-concept experiments, our architecture fixed 82 percent of the test cases. Further, an experiment with a hypothetical Apache Web server vulnerability showed that the architecture could produce a correct fix in 3 seconds, creating a total cycle (from detection to server update, minus the testing) of less than 10 seconds. Here, we offer an overview of existing approaches and our own method, as well as the benefits and challenges our approach entails.

Existing approaches

Most worm-related research falls into three categories: signature-based network filtering, proactive software protection, and containment.

Content filtering

Despite promising early results, countering worms using content filtering and coordination alone will not suffice in the future. We base this assertion on two primary observations. First, to achieve coverage, content filters are intended for use by routers, such as Cisco's network-based application recognition (NBAR). As we pointed out earlier, routers' limited packet-processing budget

makes it easy for even mildly polymorphic worms to evade filtering.[6] Network-based intrusion-detection systems (NIDS) have encountered similar problems, requiring fairly invasive packet processing and queuing at the router or firewall. Also, because they must be placed in the application's critical path, such filtering mechanisms adversely affect performance.

Second, the use of opportunistic encryption—which doesn't strictly require client or often even server-side authentication—makes it possible for worms to cover their tracks. Because it's inherently unsolvable, encryption introduces a much greater problem than filtering. Even without encryption, the increasing complexity and composability of communication protocols (such as SOAP) pose similar challenges in terms of detection and filtering.

Prevention and containment approaches

Another approach to preventing zero-day worms is to eliminate or minimize remotely exploitable vulnerabilities, such as buffer overflows. However, detecting potential buffer overflows is a difficult problem and only partial solutions exist. Blanket solutions—such as StackGuard or MemGuard[7]—typically either:

- reduce system performance, or
- detect overflows after they've occurred and overflowed stack information, thus making continuation undesirable.

They are therefore inappropriate for high-performance, high-availability environments such as a heavily used e-commerce Web server. Type-safe retrofitting of code and fault isolation techniques incur similar performance degradation, or are altogether inapplicable in some cases.

Traditionally, antivirus software has also been widely used to protect against worms. The obvious problem here is that the approach primarily depends on up-to-date virus definitions, which, in turn, primarily rely on human intervention. Further, the approach's non-real-time characteristics render it highly ineffective against zero-day worms (or those that are effectively zero-day, such as "Witty," which appeared one day after the exploited vulnerability was announced). Likewise, patch management is ineffective against zero-day worms; even as a tool against well-known vulnerabilities, its use poses several challenges to administrators.[8]

Another technology, La Brea, attempts to slow the growth of Transmission Control Protocol (TCP) worms by accepting connections and then blocking them indefinitely, causing the corresponding worm thread to block also (see www.threenorth.com/LaBrea/LaBrea.txt). Unfortunately, worms can avoid these mechanisms by probing and infecting asynchronously. With connection-throttling,[9] each host restricts the rate at which connections can be initiated. If adopted universally, such an approach could reduce a worm's spreading rate by up to an order of magnitude without affecting legitimate communications.

Automatic patching: An overview

We propose an end-point first-reaction mechanism that tries to automatically patch vulnerable software by identifying and transforming the code surrounding the exploited flaw. Our approach focuses on memory violations, specifically buffer overflows, as they continue to constitute the vast majority of *infection vectors*—byte streams that cause the worm to appear in the target system if they're "fed" to the target application.

How it works

We use instrumented versions of an enterprise's important services—such as Web servers—in a sandboxed environment. This environment is operated in parallel with the production servers, and doesn't 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 (that is, without necessarily blocking requests). A request that causes a buffer overflow on the working production server will have the same effect on the sandboxed version. Appropriate instrumentation lets us determine the buffers and functions involved in a buffer overflow attack. We then apply several source code transformation heuristics aimed at containing the buffer overflow.

Using the same sandboxed environment, we test patches against both the infection vectors and a site-specific functionality test suite to identify any obvious problems. We could also add a comprehensive test suite that would possibly take several minutes to complete, at the expense of quick reaction (although we could block access to the vulnerable service during testing using a firewall rule). Even at that, our approach is orders of magnitude faster than waiting for a "proper" fix.

If the tests succeed, we restart the production servers with the new version of the application. We're careful to produce localized patches, so we can be fairly confident that they won't introduce additional instabilities—although we (obviously) cannot offer any guarantees; additional research is needed in this area. Also, patch generation and testing occurs in a completely decentralized and real-time fashion. We therefore don't require a centralized update site, which can itself become an attack target (as was the intent of the W32/Blaster worm).

General assumptions

Our architecture uses several components that were developed for other purposes. Its novelty lies in combining components to fix vulnerable applications without unduly impacting their performance or availability.

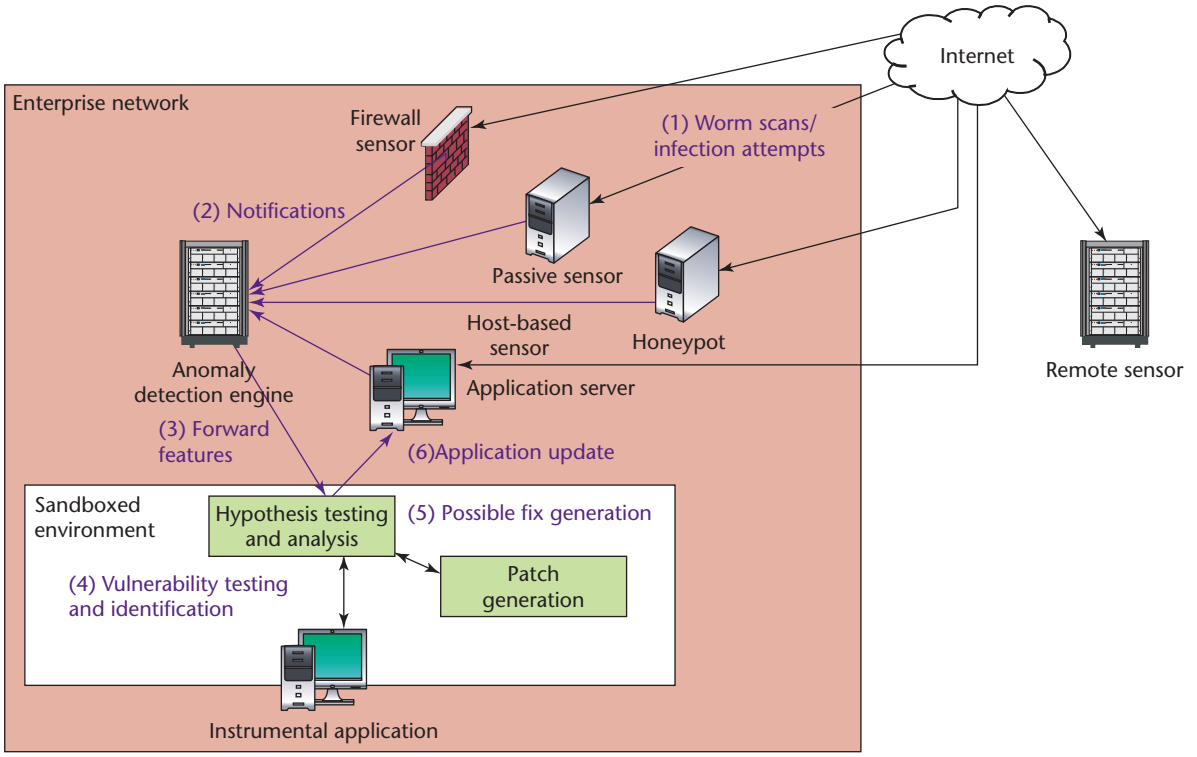


Figure 1. The worm vaccination architecture. (1) Sensors deployed at various network locations detect a potential worm and (2) notify an analysis engine, which forwards the infection vector and relevant information to a protected environment (3). The potential infection vector is tested against an appropriately instrumented version of the targeted application (4) to identify the vulnerability. (5) Several software patches are generated and tested using several different heuristics. If a patch is not susceptible to the infection and doesn't impact functionality (6), the system updates the main application.

Our major assumption is that we can extract a worm's infection vector—or one instance of it, for polymorphic worms—using various mechanisms such as honeypots and host- and network-based intrusion-detection sensors. Vector extraction is a necessary precondition to any reactive or filtering-based solution to the worm problem. The Anomalous Payload-based Network Intrusion Detection system[10] is one example of a likely-vector identification mechanism.

A secondary assumption is that the application's source code is available. Although our architecture can use binary rewriting techniques, we focus here on source code transformations. Several popular server applications are open source, including Apache, Sendmail, MySQL, and Bind. Furthermore, although our architecture can be used, as is, to react to lower-intensity "hack-in" attempts, we focus here on the more high-profile (and easier to detect) worm problem.

Benefits

Although it takes a few seconds to create and test a patch, throttling a worm's infection rate can begin immediately after a vulnerability is detected, either by halting the production server while the vulnerability is being patched or by setting a firewall rule that drops all production server traffic. When throttling and patching are incorporated into the logistic equation[5]—that is, a model used to analyze worm propagation—our approach clearly helps decelerate a worm's initial spread.

System architecture

As Figure 1 shows, our worm vaccination architecture uses

- a set of worm-detection sensors;
- a correlation engine;
- a sandboxed environment running instrumented versions of the enterprise network applications, such as the Apache Web server and the mySQL database server;
- an analysis and patch-generation engine; and
- a software update component.

Worm detection and the correlation engine

The worm-detection sensors detect potential worm probes and infection attempts. To this end, our architecture concurrently employs several sensors:

- Host-based sensors monitor the behavior of applications and servers.
- Passive sensors on the corporate firewall or on independent boxes eavesdrop on traffic to and from the servers.
- Honeypots simulate the target application's behavior and capture any communication.

We also use other types of sensors, including NIDS, intrusion prevention systems (IPSs), and similar sensors run by other entities. The system can use any combination of sensors simultaneously; honeypot servers are the most promising because worm probes don't distinguish between real and fake servers. Honeypots and other deception-based mechanisms are also effective against hit-list-based worm propagation strategies, assuming they're in place during the scanning phase.[5]

The sensors communicate with each other and with a central server, which correlates events from independent sensors and determines potential infection vectors (such as HTTP request data, as seen by a honeypot). Independent entities can then, in turn, use these infection vectors to test their applications for susceptibility. The payload-based anomaly detector (PAYL),[10] is an example of a likely-vector identification mechanism. PAYL models the network traffic's normal application payload 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. This method has multiple advantages: it's stateless, doesn't parse the input stream, and generates a small model that can be updated using an incremental online learning algorithm to maintain an accurate real-time model. Real-time mechanisms such as PAYL are important because they also provide a way to potentially identify Warhol (hit-list) worms.

The correlation engine determines the likelihood of any particular communication being an infection vector (or a manually launched attack) and requests sandbox testing for suspicious communications. It also maintains a list of fixed exploits and false positives (communications that have already been deemed innocuous or at least ineffective against the targeted application).

Sandboxed environment

Because we use the instrumented server only for clean-room testing, the instrumentation for target applications can be fairly invasive in terms of performance. The performance degradation from using a mechanism like ProPolice or Dynamic Buffer Overflow Containment (DYBOC),[11] our vulnerability detection tool, can introduce up to an order of magnitude increase in overhead. Although this is prohibitive in production systems, it's acceptable for testing purposes.

At its most powerful, the test environment could use a full-blown machine emulator to determine whether the application has been subverted. Other potential instrumentation includes lightweight virtual machines, dynamic analysis/sandboxing tools, or mechanisms such as MemGuard. Such mechanisms are generally not used for application protection because of their considerable impact on performance. They also typically cause the application to fault and cease operation. In our approach, this drawback is not particularly important given the sandbox context and our goal of identifying, as accurately as possible, the source of the application's weakness. MemGuard,[7] for example, can identify both the specific buffer and function that are exploited in a heap-overflow attack. Alternatively, by using an emulator, 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 more likely it is to detect subversion and identify the vulnerability's source. The analysis step itself can only identify known attack classes, such as a stack-based buffer overflow. However, new attack classes appear less often than specific attack exploits.

Patch generation and testing

Armed with knowledge of the vulnerability, we can automatically patch the program. Generally, program analysis is an impossible problem. However, there are a few fixes that might mitigate an attack's effects.

The most promising approach—and thus our primary focus—is to move 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 allocated buffer's size so that it's larger than that of the infection vector, thus protecting the application from even crashing (for fixed-size exploits). We can then use a version of `malloc()` to allocate two additional write-protected pages to bracket the target buffer. Any buffer overflow or underflow will send a segmentation violation (SEGV) signal. A signal handler that we added to the source code catches the SEGV signal, and can then `longjmp()` to the code immediately following the overflow-causing routine. Although we could make this a blanket approach—applying it anywhere in the code where a buffer overflow might occur—that would entail a significant performance impact. Instead, we simplify the problem somewhat, using the worm's attack side effects as a hint to the potential vulnerability's location.

There are also other possible approaches to mitigating an attack's effects, including:

- We can use minor code-randomization techniques[12] to "move" the vulnerability so that the infection vector no longer works.
- We can attempt to "slice-off" some functionality by

immediately returning from the mostly unused code that contains the vulnerability. For large software systems with numerous, often untested, features that are not regularly used, this might be the solution with the least impact. To determine whether functionality is unused, we can profile the real application; if the vulnerability is in an unused application section, we can logically remove that part of the functionality (with an early function-return, for example).
- We can add code that recognizes either the attack itself or specific conditions in the stack trace (such as a specific stack records sequence), and then returns from the function if it detects these conditions. In a sense, this approach is equivalent to content filtering, and least likely to work against even mildly polymorphic worms.

We plan to investigate additional heuristics in future research.

To avoid introducing application instability, we use localized patches. Although it's difficult, if not impossible, to argue about the correctness of any newly introduced code (whether created by humans or automated processes such as our own), we're confident that our patches don't exacerbate the problem for two reasons. First, the patches have minimal scope. Second, they emulate behavior that a compiler or other automated tool could have introduced automatically during the code authoring or compilation phase. Although this is by no means a proof of correctness, we believe it is a good argument for our method's safety.

Our architecture lets us easily add new analysis techniques and patch-generation components. We can test several patches (potentially in parallel) until we're satisfied that the application is no longer vulnerable to the specific exploit. To ensure that the patched version will continue to function, we use a site-specific test suite to determine what functionality (if any) has been lost. The test suite is generated by the administrator in advance, and should reflect a typical application workload, exercising all critical aspects (such as performing purchasing transactions for an e-commerce site). Our system also lets us proactively patch and test all possibly vulnerable code locations to determine, prior to an attack, whether any undesirable side effects might manifest. Naturally, if no heuristic works, it's impossible to automatically fix the application and other measures must be used.

Application update

Once we have a worm-resistant application version, we must instantiate it on the server. Thus, our application's final component 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 application's new version, terminates the running instance (gracefully, if possible), replaces the executable with the new version, and restarts the server.

Implementation

Our prototype implementation is comprised of

- ProPolice, to identify software vulnerabilities;
- TXL, to apply potential patches; and
- a sandboxed environment, to provide a secure environment.

ProPolice

To detect the source of buffer overflow/underflow vulnerabilities, we use the OpenBSD version of ProPolice (www.trl.ibm.com/projectes/security/spp/). ProPolice is a Gnu Compiler Collection (GCC) extension for protecting applications from stack-smashing attacks in a manner similar to StackGuard.[7] ProPolice returns the names of the function and offending buffer that lead to the overflow/underflow condition. It then forwards this information to TXL.

ProPolice has several protection features:

- It reorders local variables to place buffers after pointers. This reordering prevents the corruption of pointers that the attack could use to further corrupt arbitrary memory locations,
- It copies function argument pointers to an area preceding local variable buffers. This also prevents pointer corruption, which can further corrupt arbitrary memory locations.
- It omits instrumentation code from some functions to decrease performance overhead.

When an attacker attempts a buffer overflow attack on applications compiled with ProPolice extensions, the program execution is interrupted and it reports the offending function and buffer. ProPolice incurs a modest performance overhead, similar to StackGuard's.[7] Note that the application under attack detects overflows after they've occurred—and have already overflowed stack information—making program continuation undesirable. While this is more palatable than outright subversion, it is suboptimal in terms of service availability.

Although ProPolice was sufficient for our prototype implementation, a fully functional system would use a better mechanism, such as Valgrind (http://valgrind.kde.org) or MemGuard.[7] Both systems can catch all illegal memory-dereferences (even those in the heap). They're also considerably slower than ProPolice and can slow down an application by up to an order of magnitude. This makes them unsuitable for production systems, but usable in our system where performance is less relevant.

TXL

Armed with information produced by ProPolice, our system invokes TXL[13] to transform the code. TXL is a hybrid functional and rule-based language that's well suited for source-to-source transformation and for rapidly prototyping new languages and language processors. It specifies the grammar responsible for parsing the source input in a notation similar to Extended Backus-Nauer (BNF). TXL also supports several parsing strategies, making it comfortable with ambiguous grammars. This accommodates more natural, user-oriented grammars and thus circumvents the need for strict compiler-style implementation grammars.

In our system, TXL performs C-to-C transformations by changing the American National Standards Institute (ANSI) C grammar. In particular, we move to the heap variables originally defined on the stack using a simple TXL program.[11] The script examines source declarations and transforms 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. The other heuristic we use is a "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. As Figure 2 shows, protected malloc (`pmalloc()`) allocates two additional, zero-filled write-protected memory pages that surround the requested allocated memory region. Any buffer overflow/underflow will cause the operating system to issue a SEGV signal to the process. We use `mprotect()` to mark the surrounding pages as read-only.

Our TXL script inserts a `setjmp()` call immediately before the function call that caused the buffer overflow. This operation saves the stack pointers, registers, and program counter so that the program can later restore their state. We also inject a signal handler that catches the SEGV signal and calls `longjmp()`, restoring the stack pointers and registers (including the program counter) to their values prior to the call to the vulnerable function (in fact, they are restored to their values as of the call to `setjmp()`). The program will then reevaluate the injected conditional statement that includes the `setjmp()` call. This time, however, the return value causes the conditional to evaluate to false, thereby skipping the offending function's execution. 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:

- Heap objects are protected from being overwritten by an attack on the specified variable, because there is a signal violation when data is written beyond the allocated space.
- Because we can recover the stack context environment prior to the offending function's call, we can recover gracefully from an overflow attempt.

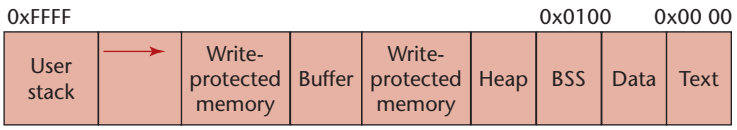


Figure 2. Protected malloc. Write-protected memory pages surround a buffer allocated with `pmalloc()`.

Examining the source code of the Code Security Analysis Kit (CoSAK) project's programs showed that most calls that caused an overflow/underflow—such as `strcpy()` and `memcpy()`—didn't check for return values or include calls to other routines. This is an important observation, because it validates our assumption that the heuristic can circumvent the malignant call using `longjmp()`.

Sandboxed environment

Finally, for our sandboxed environment, we run the OpenBSD operating system on the VMWare virtual machine. VMWare lets developers isolate operating systems and software applications from the underlying OS in secure virtual machines that coexist on a single piece of hardware. Once we have created a correct application version, we simply update its image on the production environment outside the virtual environment, and restart it.

Experimental evaluation

To illustrate our system's capabilities and the patch heuristics' effectiveness, we constructed a simple file-serving application that had:

- a buffer overflow vulnerability, to test against stack-smashing attacks; and
- superfluous services, to test slice-off functionality.

The application used a simple, two-phase protocol in which a service is requested (through different functions in the code), and then the application waits for the next request from the network. We wrote the application in ANSI C. We constructed a buffer overflow attack to overwrite the return address and attempt to get access to a root shell. We compiled the application under OpenBSD, with the ProPolice extensions to GCC.

Once ProPolice provided the names of the function and buffer potentially responsible for the buffer overflow, we invoked the TXL implementation of our heuristics. At that point, we tested the heuristics and recompiled the

TXL-transformed code, then ran a simple functionality test on the application. The test—on whether the application could correctly serve a given file—was a simple script that attempted to access the available service. This application was an initial proof-of-concept for our system, and didn't prove our approach's correctness. We acquired more substantial results by examining the CoSAK project's applications.

CoSAK data

To further test our heuristics, we examined several vulnerable, open-source software products through CoSAK, run by Drexel University's software engineering research group (see http://serg.cs.drexel.edu/cosak/index.shtml). 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. The team has compiled a database of 30 open-source software products along with their known vulnerabilities and respective patches. The database is comprised of tools, products, and libraries with known vulnerabilities, many of which are listed as susceptible to buffer overflow attacks.

When we tested the move-to-heap heuristic against the CoSAK data set, our system fixed 14 out of 17 buffer overflow vulnerabilities (an 82 percent success rate). The remaining 13 products were not tested because their vulnerabilities were unrelated to buffer overflow. We examined the three products that our solution didn't fix, and in all cases, an appropriate fix would require adjustments to the TXL heuristics to cover special cases.

Calls to the `strcpy()` routine caused most CoSAK dataset vulnerabilities. Examining the respective security patches showed that, for most cases, the buffer overflow susceptibility could be repaired by a respective `strncpy()`. Furthermore, most routines didn't check for return values and didn't include routines within the routines, thus providing fertile ground for our `pmalloc()` heuristic.

Performance

To evaluate our system's performance, we tested the patch-generation engine on an instrumented version of Apache 2.0.48. We chose Apache due to its popularity and source code availability. We tested basic Apache functionality, omitting additional modules. The evaluation's purpose was to validate the hypothesis that heuristics can be applied and tested in a timely manner. We conducted the tests on a PC with an AMD Athlon processor operating at 2.8 GHz and 2 GBytes 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 a patch heuristic wouldn't require a complete application recompilation. To get a realistic insight on the time required from applying a patch to starting the application test, we applied our move-to-heap TXL transformation on many different files, ranging from large to small sizes, and recompiled Apache's latest version. The average time across the different files for compilation and relinking was 2.5 seconds.

Another important performance issue is the TXL transformation time for our basic heuristics. Our ability to pass the specific function name and buffer to TXL greatly reduces the transformation time because the ruleset is concentrated on a targeted section of the source code. The average transformation time for different examined functions was 0.045 seconds. This result is very encouraging; it lets us assume that we can apply the heuristics in under 3 seconds.

Discussion

Our work here is in its preliminary stages, and naturally creates many questions as to its practicality, use, and safety.

Challenges

There are several challenges associated with our approach, including attack identification, reliable software repair, source code availability, and multipartite worms.

Attack identification. First, our approach must determine the attack's nature (such as a buffer overflow) and identify likely software flaws that permit the exploit. Obviously, our approach can only fix known classes of attacks, such as stack or heap-based buffer overflows. This knowledge manifests itself through the debugging and instrumentation of the application's sandboxed version. However, we don't need to know of the specific instances of such attacks; this is what we designed the system itself to determine.

Currently, ProPolice identifies the likely functions and buffers that lead to the overflow condition. Our architecture could use more powerful analysis tools to catch more sophisticated code-injection attacks; we intend to investigate them in future work. Furthermore, though we've yet to investigate it, our architecture should be general enough to detect other attack classes, such as email worms.

Reliable software repair. As we noted earlier, reliability in this area is impossible to guarantee. Our heuristics let us generate potential fixes for several buffer overflow classes using code-to-code transformations,[13] and test them in a clean-room environment. We must further research automated software recovery to develop better repair mechanisms.

Among our research plans are investigating the possible use of Aspect-Oriented Programming[14] to create source code locations ("hooks") that would let us insert appropriate fixes. When the appropriate repair model is plugged in, our architecture can automatically fix any

software fault type, including invalid memory dereferences.[15] When it's impossible to automatically obtain a software fix, we can use "content filtering"[16] to temporarily protect the service. Combining our approach with content filtering is another topic of future research.

Source code availability. Our system assumes that a target application's source code is available, making patches easy to generate and test. When that's not the case, we could apply binary-rewriting techniques[17]—at considerably higher complexity. Application instrumentation also becomes correspondingly more difficult under some schemes. One intriguing possibility would be if vendors ship two application versions: one regular, and one instrumented. The latter could then provide a standardized set of hooks that permit oversight by a general monitoring module.

Multipartite worms. Multipartite worms use multiple, independent infection vectors and propagation mechanisms, such as spreading over both email and HTTP. Our architecture treats such infections as independent worms.

Centralized vs. distributed reaction

Some researchers 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.[5] However, it seems unlikely that users would trust an entity other than the software vendor to arbitrarily patch software. Furthermore, people still need to craft the fixes, and they'd thus arrive too late to contain worms.

In our scheme, a CCDC-type enterprise would serve as a real-time alert coordination and distribution system. Individual enterprises would 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, the CCDC could use our architecture to distribute "fake worms." The system would treat this channel as another sensor supporting the analysis engine. Fake-worm propagation would trigger the creation of a quick fix if the architecture deemed the worm bona fide (that is, if the application crashed when the attack was run in the sandbox). Again, this would serve as a mechanism for distributing quick patches by independent parties, by distributing only the exploit and letting organizations create their own patches.

Although our speculations assume that such a system will be deployed in every medium to large enterprise network, there's nothing to preclude resource pooling 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 security company would then push fixes to all clients. Some managed security vendors take a similar approach, keeping programmers available on a 24-hour basis. In all cases, administrators must be aware of the services offered (officially or unofficially) by all of their network hosts.

Attacks against the system

Naturally, our system should avoid giving attackers new opportunities to subvert applications and hosts. One concern is the possibility of attackers "gaming," as when they opportunistically attack systems solely to cause instability and unnecessary software updates.

One interesting attack would be to cause oscillation between software versions that are alternatively vulnerable to different attacks. Although this might be theoretically possible, we can't think of a suitable example. Such attack capabilities are limited by the fact that the system can test the patching results against current attacks, as well as previous ones that are still pending (not officially fixed by an administrator-applied patch). Furthermore, we assume that the various system components are appropriately protected against subversion—that is, that the clean-room environment is firewalled, using TLS/SSL or IPsec to protect component communications and interactions.

If attackers subvert a sensor and use it to generate false alarms, event correlation will reveal the anomalous behavior. In any case, the sensor can at best mount only a denial-of-service (DoS) attack against the patching mechanism by causing it to test many hypotheses. Again, such anomalous behavior is easy to detect and counter, without impacting either the protected services or the patching mechanism.

Our architecture could also be attacked using a DoS to deny communication between the correlation engine, the sensors, and the sandbox. Such an attack might in fact be a by-product of a worm's aggressive propagation, as with the SQL worm (see www.siliconedefense.com/research/worms/slammer). Fortunately, it should be possible to filter the ports used for these communications, making it difficult to mount such an attack from an external network.

Individual enterprises would independently confirm the validity of a reported weakness and create their own fixes in a decentralized manner.

As with any fully automated task, we don't yet fully understand the risks of relying on automated patching and testing as the only real-time verification techniques. To the extent that our system correctly determines that a buffer overflow attack is possible, the system's operation is safe: it will either create a correct patch for the application, or the application will be shut-down or replaced with a non-working version. Considering the alternative—guaranteed service loss and application subversion—we believe that many people will find the risk acceptable. The question then focuses on the analysis engine's correctness. This appears to be a fundamentally impossible problem—our architecture enables us to add appropriate checks as needed, but we cannot guarantee absolute safety. As with all engineering endeavors, we seek to determine whether our system presents an acceptable tradeoff in certain environments.

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 CCDC. 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 several vulnerable applications as hypothetical worm-infection targets shows promising results that validate our approach and will spur further research. ▢

Acknowledgments

We thank the anonymous reviewers for their constructive comments and guidance during manuscript preparation.

References

1. D. Moore, C. Shanning, and K. Claffy, "Code-Red: A Case Study on the Spread and Victims of an Internet Worm," *Proc. 2nd Internet Measurement Workshop* (IMW), ACM Press, 2002, pp. 273–284.
2. C.C. Zou, W. Gong, and D. Towsley, "Code Red Worm Propagation Modeling and Analysis," *Proc. 9th ACM Conf. on Computer and Comm. Security* (CCS), ACM Press, 2002, pp. 138–147.
3. E. Levy, "Crossover: Online Pests Plaguing the Offline World," *IEEE Security & Privacy*, vol. 1, no. 6, 2003, pp. 71–73.
4. D. Moore et al., "Internet Quarantine: Requirements for Containing Self-Propagating Code," *Proc. IEEE Infocom Conference,* IEEE Press, 2003; http://www.ieee-infocom.org/2003/papers/46_04.PDF.
5. S. Staniford, V. Paxson, and N. Weaver, "How to Own the Internet in Your Spare Time," *Proc. 11th Usenix Security Symp.*, Usenix Assoc., 2002, pp. 149–167.
6. M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 169–186.
7. C. Cowan et al., "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. 7th Usenix Security Symp.*, Usenix Assoc., 1998, pp. 63–78.
8. J. Twycross and M.M. Williamson, "Implementing and Testing a Virus Throttle," *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 285–294.
9. E. Rescorla, "Security Holes...Who Cares?" *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 79–90.
10. K. Wang and S. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," *Proc. 7th Int'l Symp. Recent Advances in Intrusion Detection* (RAID), LNCS 3224, Springer-Verlag, 2004, pp. 201–222.
11. S. Sidiroglou, G. Giovanidis, and A.D. Keromytis, "A Dynamic Mechanism for Recovering from Buffer Overflow Attacks," *Proc. 8th Information Security Conf.* (ISC), LNCS 3224, Springer-Verlag, 2005, pp. 1–15.
12. S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 105–120.
13. A.J. Malton, "The Denotational Semantics of a Functional Tree-Manipulation Language," *Computer Languages*, vol. 19, no. 3, 1993, pp. 157–168.
14. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming*, vol. 1241, M. Aksit and S. Matsuoka, eds., Springer-Verlag, 1997, pp. 220–242.
15. S. Sidiroglou et al., "Building A Reactive Immune System for Software Services," *Proc. Usenix Annual Technical Conf.*, Usenix Assoc., 2005, pp. 149-161.
16. J.C. Reynolds et al., "The Design and Implementation of an Intrusion Tolerant System," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN), IEEE CS Press, 2002, pp. 285–292.
17. M. Prasad and T. Chiueh, "A Binary Rewriting Defense Against Stack-Based Buffer Overflow Attacks," *Proc. Usenix Annual Tech. Conf.*, Usenix Assoc., 2003, pp. 211–224.

Angelos Keromytis *is an assistant professor in the Department of Computer Science at Columbia University, where he directs the Network Security Lab. His research interests include survivable systems, network security, cryptographic protocol design and analysis, and operating systems. He has a PhD in computer science from the University of Pennsylvania. He is a member of the ACM, IEEE, and USENIX. Contact him at angelos@cs.columbia.edu.*

Stelios Sidiroglou *is a PhD candidate in the Department of Computer Science at Columbia University, where he is a member of the Network Security Laboratory. His research interests include operating systems, network security, and survivable software. He has an MS in electrical engineering from Columbia University. Contact him at stelios@cs.columbia.edu.*