Bezoar*: Automated Virtual Machine-based Full-System Recovery from Control-Flow Hijacking Attacks

Daniela A. S. de Oliveira Jedidiah R. Crandall Gary Wassermann Shaozhi Ye
S. Felix Wu Zhendong Su Frederic T. Chong
University of California, {Davis, Santa Barbara} University of New Mexico
{oliveira,wassermg,yeshao,wu,su}@cs.ucdavis.edu chong@cs.ucsb.edu crandall@cs.unm.edu

***Abstract*—System availability is difficult for systems to maintain in the face of Internet worms. Large systems have vulnerabilities, and if a system attempts to continue operation after an attack, it may not behave properly. Traditional mechanisms for detecting attacks disrupt service and current recovery approaches are application-based and cannot guarantee recovery in the face of exploits that corrupt the kernel, involve multiple processes or target multithreaded network services. This paper presents Bezoar, an automated full-system virtual machine-based approach to recover from zero-day control-flow hijacking attacks. Bezoar tracks down the source of network bytes in the system and after an attack, replays the checkpointed run while ignoring inputs from the malicious source. We evaluated our proof-of-concept prototype on six notorious exploits for Linux and Windows. In all cases, it recovered the full system state and resumed execution. Bezoar incurs low overhead to the virtual machine: less than 1% for the recovery and log components and approximately 1.4X for the memory monitor component that tracks down network bytes, for five SPEC INT 2000 benchmarks.**

I. Introduction

System availability is difficult for online systems to maintain in the face of Internet worms. Control-flow hijacking worms perform their attacks by overwriting control data in a victim host, which allows them to perform arbitrary malicious actions. Software systems cannot be guaranteed to be free from vulnerabilities because designers and programmers do make mistakes and current verification and testing techniques cannot assure that a piece of software meets its specification in the presence of errors or bad inputs. Thus, techniques for software fault-tolerance are necessary to guarantee availability in real-world systems.

Fault-tolerance is comprised of four phases [26]: (i) error detection, (ii) damage confinement and assessment, (iii) error recovery, and (iv) fault treatment and continued system service. In the context of control-flow hijacking attacks, defense systems usually address (i) by detecting and stopping the exploit using information flow tracking [14], [30], [33], [35], address-space randomization [6], [50], memory protection [24], [28] and other methods [13]. Usually, after attack detection, the system crashes or reboots, or the offending application is terminated, which causes disruption of service. For example, if an IDS (intrusion detection system) reboots the system, it will drop non-malicious clients, lose recently acquired data, and experience down-time as it restarts and replenishes cached data for services like DNS. Even if the system continues its execution after the attack (by killing the offending application, for instance), it could not be able to proceed. The exploit could have corrupted areas of memory used by non-malicious processes and the system could eventually crash or hang. Also, the attacker could have damaged critical system files or data structures and the continuation of the host execution could lead to a situation of error or inconsistency. Some defense systems perform (ii) in varying degrees by analyzing an attack once it is stopped and patching the vulnerability with filters or signatures [9], [12], [15], [27], [47]. More recently the research literature proposed some techniques to address (iii) [36]–[40], [44]. Post-attack recovery capabilities when applied to the hosts in a managed network have a great impact in the whole management strategy for security because it improves dependability of the hosts. Most of the recovery approaches proposed are application-based [36], [38]–[40], [44], *i.e.*, they aim at recovering a specific process (usually the victim one) after a worm attack.

Application-based recovery approaches cannot guarantee recovery from the new generation of attacks. Kernel exploits are expected to be common in the near future [15], [23]; an exploit does not necessarily need to involve an user-space process because a vulnerability in the kernel will allow the execution of malicious code in supervisory mode. Further, certain OS's, such as Windows, handle network traffic in the kernel and certain exploits, such as heap overflows [25], modify values within the heap management information structure and corrupt memory outside the victim process address space. Also, an exploit may involve multiple processes, multithreaded services, and use multiple stages (*e.g., innd* exploit [1]). In a multithreaded Web server, for instance, the malicious network packet can be first received by a thread listening on a certain port, but the memory corruption itself may only occur in another thread, spawned just to treat the newly arrived TCP connection. For an application-based approach it may not be trivial to decide which thread needs to be

*Bezoar is a stone taken from the stomach of ruminant animals that was formerly believed to be an antidote from most poisons.

This work has been supported by grants FA9550-07-1-0532 (AFOSR MURI) and 0335299, 0520269, 0627749 (NSF).

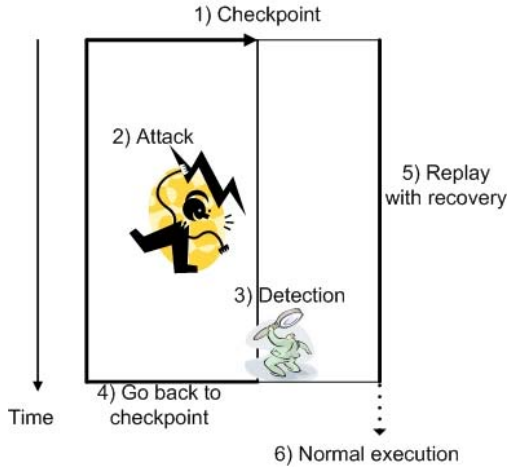

Fig. 1. Bezoar

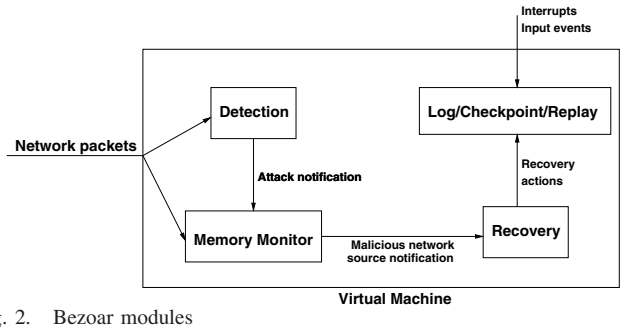

Fig. 2. Bezoar modules

recovered: the one that first received the malicious network input or the one that actually caused memory corruption? Other limitations of current recovery approaches include: not recovering damage to the file system; losing state related to interprocess communication, messages, signals, and resources (*e.g.*, files opened, processes spawned and pages allocated); undoing the effects of the attack speculatively [36], [38], [39] and requiring the application source code [36], [40].

This paper presents an automated full-system virtual machine-based approach to recover from zero-day (*i.e.*, that exposes an unknown vulnerability) control-flow hijacking attacks, implemented as Bezoar. It can recover the full-system state (processes and kernel space, resources and file system), incurs low overhead to the virtual machine (VM) and is OS and application independent. Figure 1 provides an overview of our recovery strategy. The main idea is that during normal execution the system takes periodic checkpoints while logging all architectural non-deterministic events and tracks down how data from the network propagates into the memory. In the event of an attack, we discover its source, go back to the most recent checkpoint and replay the system execution while ignoring all network packets coming from the malicious source.

We have implemented this approach as a proof-of-concept prototype using Bochs [48], a host VM [41] that emulates the IA-32 Pentium architecture. Bezoar has been successfully tested using six notorious exploits (Code Red II, Blaster, Slammer, *innd*, *rpc.statd* and *wu-ftpd*) in Windows and Linux. Our log and recovery components incur very low overhead to the VM (less than 1%) and the component that tracks down network bytes into memory incurs slowdown of approximately 1.4X.

The rest of the paper is organized as follows. Section II describes in detail Bezoar's design and implementation. In Section III we present the experiments we have performed to validate our recovery approach. Section IV discusses Bezoar issues and limitations, and Section V discusses related work. Our conclusions and future work are presented in Section VI.

II. Bezoar Design and Implementation

A. High-level View

Bezoar is an automated VM-based recovery approach composed of the following modules (Figure 2): detection, memory monitor, log/replay/checkpoint and recovery. During normal execution the system logs all architectural non-deterministic events (interrupts and input events). The memory monitor module tracks down how data from the network propagates into the memory (including registers and memory from external devices). Each memory byte will be associated with zero, or one, or more network sources. When an attack is detected, we discover its source and go back in time to the most recent checkpoint for recovery through replay.

The complete system execution is replayed until the first malicious input enters the system in a network packet. The malicious network input event is ignored (as all subsequent network packets coming from the same source) and the execution enters a semi-replay phase.

In this phase the system executes partially in normal mode (without logging) and replay mode: it accepts and processes new events, but also replays non-malicious network packets so that their effects can be propagated up to the point the attack was originally detected. When the last non-malicious network packet is replayed, the system returns to normal mode. During recovery (replay and semi-replay) the system does not send any data to the network.

B. Attack Detection

We have used our previous work Minos [14] to detect zero-day control flow hijacking attacks. It is a microarchitecture in which every 32-bit word of memory and general-purpose register are augmented with an integrity bit, which is set by the kernel when it writes data into them. This bit is set to low or high, depending on the trust the kernel has for it. Data coming from the network are usually regarded as low integrity. Any control transfer involving untrusted data is considered a vulnerability, and a hardware exception traps to the VM whenever this occurs. Minos was prototyped in Bochs and can be implemented with negligible overheads [20]. Our proof-of-concept prototype was implemented as an extension to this Minos-enabled Bochs. Figure 3 illustrates this architecture.

C. Memory Monitor

We introduce this module to keep track of how network bytes and their sources propagate into the system memory,

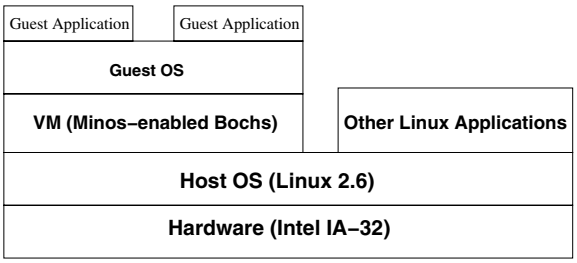

Fig. 3. VM architecture

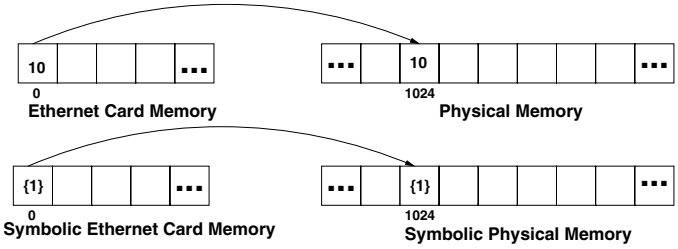

Fig. 4. Memory Monitor - Example 1

including the register bank and memory from external devices such as the network card. We make use of a symbolic memory space and a symbolic register bank to store information about the propagation of network bytes into our system. Each component of this symbolic storage area has a 1:1 correspondence with the real component in the system architecture.

The network byte propagation information is a set of integers representing the unique identification of their source. Each network source is characterized by an IP address and a port number. In our approach every time the VM Ethernet card receives a frame, we insert a new entry for the associated network source in our system if this is the first frame being received from the source. Then the frame bytes are stored into the network device memory and the identification number of their source is stored into the symbolic memory of the card at the same locations where the bytes were stored.

The memory monitor can thus keep track of how these bytes propagate into the real system memory and registers as the CPU reads, moves and processes them. Wherever these bytes (and also their derived bytes) are stored, we can find their sources at our symbolic memory space.

For example, let us suppose that byte 10 coming from network source 1 is copied from the Ethernet card to user space at memory location 1024 (Figure 4). Then the system receives

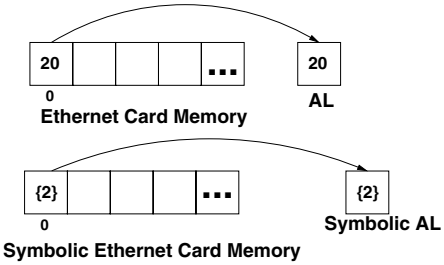

Fig. 5. Memory Monitor - Example 2

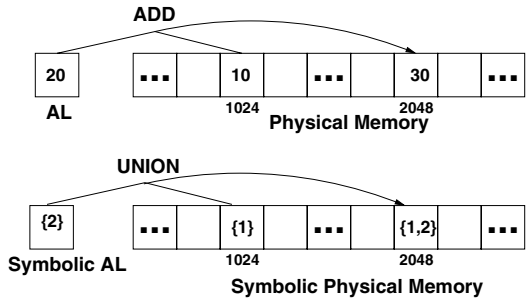

Fig. 6. Memory Monitor - Example 3

byte 20 from network source 2 which gets stored into register AL (Figure 5). If the CPU executes an instruction that adds register AL to memory location 1024, and stores the result into memory location 2048, the final state of real and symbolic memory will be as illustrated in Figure 6. Byte 30 at memory location 2048 is considered to be originated from network sources 1 and 2, and we store the union of the network source set associated with bytes 10 and 20 to the symbolic memory at location 2048. When a memory location is overwritten with a byte that is not derived from any network source, we simply remove the set of network source ID numbers that might be stored into the corresponding location in the symbolic memory.

For each network source active in the system we keep a counter for the number of symbolic memory locations storing its ID number. When this counter reaches zero, we remove the network source entry from our system. The lifespan of most network sources is very short [14], for example, data from a TCP connection will soon die in the system after the connection was served and data from new connections arrive.

D. Execution Checkpoint, Log and Replay

A post-attack recovery system needs to revert the attack actions in the victim host. We achieve this through replay, which allows us to go back in time and undo the effects of the attack in the system. We modify our previous log and replay framework ExecRecorder [32] with recovery capabilities: removing the execution of malicious events and performing the semi-replay phase. ExecRecorder has a log file growth rate of approximately 0.31 GB/hour and can replay the execution of an entire system by checkpointing the complete system state (virtual memory and CPU registers, virtual hard disk and memory of all virtual external devices) and logging all architectural nondeterministic events.

Our checkpoint strategy involves the complete system state but is lightweight because it is based on copy-on-write. It is implemented by duplicating the VM process via the *fork* system call. After the fork, the parent process waits in the background for a SIGUSR1 signal while the child process continues its execution logging events. The suspended parent represents the frozen state of the system at the time the checkpoint was taken. The checkpoint of the virtual hard disk is achieved by using an undoable disk mode [48], which is a committable/rollbackable disk image.

After the logging of a run the replay module can be called to reproduce the system execution from the checkpoint. The

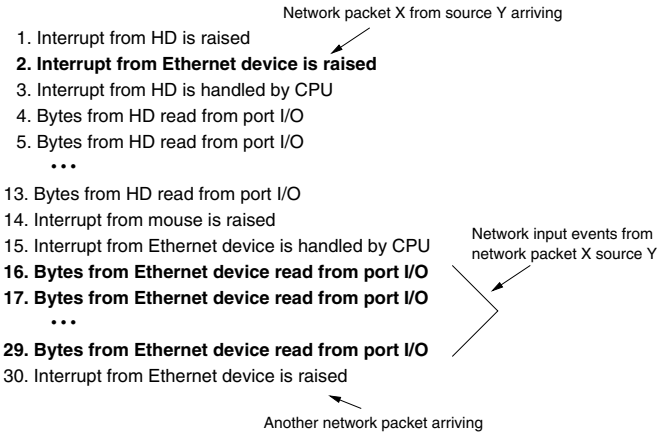

Fig. 7. Identifying the source of network input events

child process wakes up the parent process with a SIGUSR1 signal. The parent process resumes its execution using the log file to reproduce the events at the time they happened during the log phase and all interrupts or input events not prescribed in the log file are silenced.

E. Identifying Network Inputs with their Sources

Bezoar associates a network input event with its source as follows. When an Ethernet frame is read from the Ethernet device, we extract information about the source of this packet at the Network and Transport protocol levels. Bezoar maintains this information (source IP address and port number) in an object that we call **network source**. Each network source is uniquely identified by an ID (**network source ID**).

Also, at the time the network source is extracted, Bezoar keeps its ID as the current in the system until a new packet from another network source is received. After finishing receiving the frame, the Ethernet device will interrupt the CPU, signaling that there are bytes to be read through port I/O. This interrupt may not be handled immediately by the CPU because it may have to handle higher-priority interrupts. When this interrupt is actually handled, the corresponding network bytes are read through port I/O. Thus, all network input events occurring until the next Ethernet frame is received have as its network source ID the one kept as current by Bezoar. Figure 7 shows an example of this sequence of events.

F. Identifying Malicious Network Inputs

If an attack is detected during the run, Bezoar discovers its malicious network source and initiates recovery by replaying the run without processing any event originated from the malicious source. More specifically, at detection point, Bezoar will inspect in the symbolic storage area (symbolic memory and registers) the set of network sources stored at the same memory address or register from which the bogus value for register EIP came from. This memory address or register was corrupted at some point by the exploit, and the value at this location is derived from the malicious bytes the attacker sent through a network packet. In all our experiments with real worms, there was only one network source directly associated with the attack.

G. Replaying to Recover from Attacks

Bezoar starts replaying the run knowing what is the malicious network source ID. During this replay for each network input event processed, Bezoar compares the network source ID for the event with the malicious network source ID. If they are different the event is replayed because it is non-malicious. If they are the same, this means that we are seeing the first malicious input event entering the system. The malicious event is not processed and the execution enters a *semi-replay* phase.

This phase represents our solution to the *time-travel paradox* discussed in Ref. [8] where this problem is compared to the metaphor of time-traveling portrayed in a famous movie: the protagonist goes back in time to repair some action that caused bad consequences to the present. After making the necessary changes, he inevitably affects the course of other actions and when he returns to the present, he sees a "modified" present that is consequence of the action he repaired along with the inevitably side-effects on other aspects of the past.

In our particular instance of time-travel, the repair action is to not process a malicious input as if it had never happened. Up to the point that we see the first malicious input entering the system we can replay the run without worrying about side-effects. When we see the first malicious input entering the system we cannot just ignore all malicious inputs while replaying other events because events are time-dependent on one another: the tick at which they occur depends on previous events. If, during replay, we remove an event and execute other successive events, we could reach an error, *e.g.*, an interrupt being raised before data is ready. In our solution, when Bezoar sees the first malicious network input entering the system, we enter a semi-replay phase where the execution is partially in replay mode and in normal mode. Interrupts and input events not prescribed in the log file are processed, but network packets coming from the malicious source are rejected. Bezoar keeps replaying non-malicious network packets from the log file until all of these events are consumed. Our goal is to replay the system as if the malicious inputs had never occurred and bring the system state (memory, file system and processes) to a legitimate state: a state it would have reached had the attack never happened. When all non-malicious network packets in the log file are consumed, the system enters normal mode but continues to refuse any network packet coming from the malicious source. It is important to point out that the system entropy during recovery will be different from what it was during normal mode. This is because during recovery, we remove from the system all malicious events, thus affecting the values at the system entropy pool.

III. Experimental Evaluation

Our experiments aimed at verifying if Bezoar could effectively recover a system after a control-flow hijacking attack and what performance overhead it incurred to the VM. More specifically, we analyzed if Bezoar could (i) discover

the source of the attack and distinguish malicious and non-malicious events and (ii) recover the server execution along with its complete state (file system, resources opened and interprocess communication).

We have selected six notorious exploits to test our approach. These exploits represent buffer overflow or format string vulnerabilities from Linux and Windows (Table I). The Blaster worm exploits a buffer overflow in a RPC interface (DCOM vulnerability). The Slammer worm targets SQL server computers and exploits a stack buffer overflow vulnerability. Code Red II attacks the Microsoft IIS Server and exploits a buffer overflow vulnerability when a string in an HTTP GET request has its ASCII characters converted to UNICODE. The worm used for the *wu-ftpd* vulnerability was presented by Crandall *et al.* [14], and exploits an error in the file globbing functionality. The *rpc.statd* worm attacks the NFS locking mechanism, and the *innd* attack targets the innd news server.

First we have tested if a Bezoar-enabled server could continue its execution after being attacked by any one of these exploits. For each exploit we have analyzed the server in three situations: idle, executing intensively its CPU, and running a Web server in a noisy campus network. For the second situation, we have selected for Linux the UnixBench [45] benchmark and for Windows an implementation of the Sieve of Eratosthenes. For the third situation we have used the Webstone 2.5 benchmark [46] with 10 simultaneous clients fetching files of different sizes (from 500 bytes to 5 MB). The Web server running in our Linux guest OS is Apache 1.3 and in Windows is Apache 2.0. All experiments were executed on a Pentium 4 SMP with 2 3.2 GHz CPU's and 1 GB of RAM. Also, each experiment was executed three times and their results averaged.

A. Effectiveness of the Recovery

We have attacked our Bezoar-enabled server with all exploits presented in Table I and, for all of them, the server continued its execution without losing its full-system state. The microbenchmarks finalized successfully without any side-effects. In our experiments, as soon as the microbenchmark started executing, we launched the attack. Table II shows, for each exploit and three different types of workload, the number of instructions (in millions) executed from the time the malicious input entered the system through the network (attack injection) up to the time the control-flow hijacking attempt was detected.

The number of instructions executed between attack injection and attack detection determines the length of the semi-replay phase. The smaller the semi-replay phase, the less our recovery approach will cause side-effects or affect a client communicating with the server. As we have mentioned before, during semi-replay the system entropy is different from what it was during normal mode and under attack. This is because during semi-replay we remove all malicious events from the execution, thus changing the values in the system entropy pool. If during semi-replay the communication between client and server depends on newly generated random numbers, for instance, an authentication key or a TCP initial sequence number, the numbers chosen during semi-replay will be different from those chosen when the system was under attack. In this situation the remote client will keep trying to communicate with the server with the old values (*i.e.*, the values the server chose during normal mode and under attack) and this can cause a TCP connection to be reset or an authentication error.

In most of the attacks we analyzed, the size of the execution window between attack injection and attack detection is on the order of millions of instructions. In our VM, this lasts a few seconds, but for a CPU with a GHz clock speed this takes a small fraction of a second. The exceptions were the *innd* and *wu-ftpd* exploits whose semi-replay phase lasted in the order of billions of instructions when the host executed a Web server workload. This is due to the complexity of these exploits. The *innd* exploit has two steps: posting a message in a news group and then canceling it. The *wu-ftpd* exploit used in our tests is complex because it was specially developed to show the insecurity of detection techniques such as non-executable pages, return pointer protection and others [14]. It is important to point out that for the case where the system was running a Web server, our experiments were done in a local area network. In real-world attacks, that might take place across continents, it could be the case that the time between attack injection and detection lasts longer in clock time because of TCP round trip time. However, for this case we can compress the idle periods in replay by identifying the idle loops and fast-forwarding the VM's CPU tick whenever we detected that the idle loop was being executed.

We have also noticed that the semi-replay phase executes less instructions than its corresponding window during normal execution (attack injection to attack detection). This is because during semi-replay we remove malicious events from the system and, consequently, all its corresponding instructions. Also, we enter normal mode as soon as we replay the last non-malicious network packet in the log file. In pure replay there may be other events to be processed after the last network packet is consumed. Table III shows, for each exploit, the number of instructions removed from the semi-replay phase.

We have also analyzed how Bezoar affected the communication of a client and a recovery-enabled server. We have considered the Web application where the Web server suffered a remote attack while servicing requests from remote clients. We have simulated 10 simultaneous clients fetching several files of different sizes (from 500 bytes to 5 MB). Each client kept making requests to the server for one minute. In each experiment, for each one of the exploits, as soon as the clients started making their requests, we launched the attack. Table IV shows the results we have obtained (average in three experiments per case).

We have analyzed all TCP packets exchanged by clients and server to find the cause for the errors in certain connections. We have observed that there was an error every time a new connection was started during the semi-replay phase. For example, suppose the server sends a TCP segment of type SYNACK to the client during the semi-replay phase. A SYNACK segment is part of the TCP three-way handshake and carries the initial server sequence number. During semi-replay, the number chosen by the server is different from the number it chose during normal execution because the entropy

Exploit	OS	Port(s)	Class	bid [49]	Vuln. Discovery
DCOM RPC (Blaster)	Windows	135 TCP	Buffer Overflow	8205	Last Stage of Delirium
SQL Name Resolution (Slammer)	Windows	1434 UDP	Buffer Overflow	5311	David Litchfield
IIS (Code Red II)	Windows	80 TCP	Buffer Overflow	2880	eEye
wu-ftpd Format String	Linux	21 TCP	Format String	1387	tf8
rpc.statd (Ramen)	Linux	111 & 918 TCP	Format String	1480	Daniel Jacobiwitz
innd	Linux	119 TCP	Buffer Overflow	1316	Michael Zalewski

TABLE I
WORM EXPLOITS

Exploit	Idle	CPU	Web server
DCOM RPC (Blaster)	0.368	0.401	0.415
SQL Name Resolution (Slammer)	0.096	0.406	0.594
IIS (Code Red II)	18.48	20.09	21.95
wu-ftpd Format String	35.68	37.38	17630
rpc.statd (Ramen)	0.769	0.888	0.872
innd	70.3	1172	9860

TABLE II
INSTRUCTIONS BETWEEN ATTACK INJECTION AND DETECTION (IN MILLIONS)

Exploit	Instructions Removed from Semi-Replay
DCOM RPC (Blaster)	6627
SQL Name Resolution (Slammer)	65
IIS (Code Red II)	130
wu-ftpd Format String	25758
rpc.statd (Ramen)	644
innd	10145

TABLE III
NUMBER OF INSTRUCTIONS REMOVED FROM SEMI-REPLAY

Exploit	Number of Connections Tried	Number of Errors
DCOM RPC (Blaster)	7	1
SQL Name Resolution (Slammer)	7	2
IIS (Code Red II)	6	0
wu-ftpd Format String	34.75	3
rpc.statd (Ramen)	49	0
innd	70	2.25

TABLE IV
NUMBER OF CLIENT CONNECTIONS X NUMBER OF ERRORS DURING RECOVERY - AVERAGE

of the system changed during semi-replay. As Bezoar replays all non-malicious network packets, the server, after sending the SYNACK, will receive an ACK segment from the client where its ACK number reflects the initial sequence number the server chose during normal execution. The recovered server, on the other hand, sees this mismatched ACK number as an error and resets the connection.

B. Performance

Figure 8 shows the performance overhead incurred by each one of Bezoar's modules and also all modules combined for five SPEC INT 2000 benchmarks. The execution times were normalized to the execution time of the VM without any of Bezoar's modules. The log and recovery modules incur negligible overhead (less than 1%), while the memory monitor component, which tracks down the source of network bytes in the system, incurs an average slowdown of 1.4X.

The total recovery time will be slightly less than the length of the checkpoint window because the semi-replay phase executes less instructions than its corresponding phase in normal mode. In Bezoar, the recovery time is determined by the length of the checkpoint window and the smaller the checkpoint interval, the smaller the recovery time. Besides recovery time, another issue that must be taken into account when selecting a checkpoint window is the performance overhead of our checkpoint strategy. We leave to future work experiments suggesting an optimum checkpoint interval for these types of worm attacks.

IV. DISCUSSION

There are some situations in which our approach cannot guarantee recovery. Currently Bezoar always goes back to the most recent checkpoint and, depending on the length of the run, the malicious input may have entered the system earlier. Also, Bezoar currently cannot guarantee recovery if the attack involves different TCP connections, or if the memory monitor component finds more than one network source associated with the attack. Another limitation is that we currently do not recover TCP connections that start during semi-replay

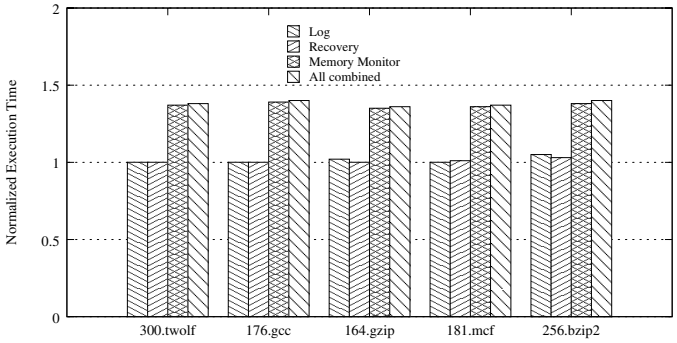

Fig. 8. Performance overhead - SPEC INT benchmarks

and fail because the entropy of the system changed during recovery. All these issues can be addressed in future work. For example, we can keep multiple checkpoints and go back several checkpoints earlier if we know which checkpoint the malicious input came from. Also, we could do recovery with multiple TCP connections by extending our current implementation to support it; the memory monitor component can already capture that more than one network source are related to a certain memory location. Bezoar can also be used against DDoS attacks if, after the first exploit, the system is patched with a vulnerability signature [9] so as other attacks will be blocked.

Although we have described Bezoar by employing the tracking of network bytes during normal mode (with 1.4X slowdown), the system can also achieve recovery by enabling the memory monitor component only after an attack has been detected. This would require two replays of the checkpointed run. The first would run tracking the source of network bytes to discover the malicious source and the second would drop malicious packets and perform the semi-replay phase. This configuration doubles the recovery time. Depending on the length of the checkpoint it may be advantageous to avoid the 1.4X slowdown in execution time during normal mode.

V. Related Work

Recovery has been addressed for a long time in areas such as databases and fault-tolerance. The approaches used for databases aim to protect data integrity [5]. In the area of software fault-tolerance the goals are to restart an application after a fault, or periodically rejuvenate a system to avoid a fault, or roll-back and/or replay an application after a fault. This can be achieved through reboot [11], [19], [42] and its variations [3], [10], software rejuvenation [22], and roll-back recovery and replay techniques [2], [7], [16], [17], [29]. These approaches cannot be directly applied for post-attack recovery because or they do not address availability, or may lose system state while recovering, or do not include a repair mechanism.

A. Recovery from General Software Failures

In Rx [34] a program is rolled back and re-executed (possibly multiple times) in a modified environment when a bug is detected. As the set of changes applied is tentative it cannot guarantee recovery. Brown and Patterson [8] propose the 3R's (Rewind, Repair and Replay) model to address the problem of recovery from mistakes made by humans. It is not automated as it requires a human operator to detect and repair an error. BackDoor [43] detects when an OS is unresponsive and salvage critical software state in the OS memory so that another recovery node executing the same application can take over the client sessions serviced by the failure node. The goal is not to recover the node after failure but to continue servicing its client sessions in spite of the failure. Other related works are the runtime systems DieHard [4] that tolerates memory errors, and Exterminator [31] that detects and corrects heap-based memory errors.

B. Recovery from Worms and Malware

Sweeper [44] is an application-based approach that extends Rx with recovery capabilities and employs heavyweight post attack analysis using multiple replays to discover the source of an attack and to generate a filter for it. INDRA [37] is an architecture that asymmetrically configures the processor cores in a security hierarchy. High privileged cores isolated from the network monitor low privilege cores running network services. It does not recover the full-system state such as file system, interprocess communication and resources (files opened, processes spawned and pages allocated). Taser [18] focus on recovering the file system after an attack using taint tracing. Sidiroglou *et al.* [38] use rescue points to restore the program execution after a fault and STEM [39] provides recovery by having the faulty application returns an error. By performing repair actions speculatively, these last two approaches cannot guarantee that the recovered application will behave properly. DIRA [40] and failure oblivious computing [36] represent compiler solutions for the problem and require the application source code. Back to the Future [21] detects an integrity violation when a trusted process is about to read untrusted data and depends on a human to classify an application as trusted and untrusted.

C. Post-Attack Analysis

DACODA [15] analyzes attacks using full-system symbolic execution in a VM. Xu *et al.* [47] and COVERS [27] analyze the victim host memory and correlate attacks to network inputs to perform signature generation. Brumley *et al.* [9] analyze the semantic of a program to generate a filter for a certain vulnerability. Vigilante [12] is a containment approach where a set of hosts run instrumented software and, upon an attack, broadcast self-certifying alerts to other hosts.

VI. Conclusions and Future Work

In this work we presented Bezoar, an automated full-system VM-based approach to recover from zero-day control-flow hijacking attacks. It monitors the propagation of network bytes in the system and, after a control-flow hijacking attack has been detected, replays the checkpointed run while ignoring inputs from the malicious source.

Our approach is promising because for all exploits analyzed we could discover the source of the attack. The execution

and the complete system state of the Bezoar-enabled server were recovered for all cases, and the communication between the server and any non-malicious client during the attack was recovered for most TCP connections. Only the connections that depended on the entropy of the system being the same as it was when the system was under attack could not be recovered. Bezoar incurs little overhead to the VM: less than 1% for the log and recovery components and an average of 1.4X slowdown for the memory monitor component that tracks down network bytes in the system. We expect that implementing Bezoar in a low-overhead VM will make it practical for use in real, online systems. As future work we plan to investigate how Bezoar modules could be implemented in hardware, and to correlate different attack sources when an exploit is multi-staged and uses different TCP connections. We also plan to implement a TCP translation scheme to avoid the entropy-related problem we experienced when a Bezoar-enabled server had to choose an initial sequence number for a TCP connection during the semi-replay phase.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments and the developers of the Bochs project.

REFERENCES

[1] Security Focus Vulnerability Notes, bugtraq id 1316.
[2] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University, 1996.
[3] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. *USENIX*, 1992.
[4] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. *PLDI*, pages 158–168, June 2006.
[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Springer-Verlag, 1987.
[6] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficent approach to combat a broad range of memory error exploits. *USENIX Security*, 2003.
[7] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. *ACM TOCS*, 14(1):80–107, February 1996.
[8] A. B. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *ACM SIGOPS European Workshop*, 2002.
[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. *IEEE Symposium on Security and Privacy*, May 2006.
[10] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. *OSDI*, 2004.
[11] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.
[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *ACM SOSP*, 2005.
[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, pages 63–78, Jan 1998.
[14] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *MICRO*, December 2004.
[15] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *ACM CCS*, pages 235–248, November 2005.
[16] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. *NDSS*, February 2006.
[17] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *University of Michigan Technical Report CSE-TR-410*, September 2002.
[18] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. *ACM SOSP*, pages 163–176, 2005.
[19] J. Gray. Why do computers stop and what can be done about it? *5th Symp. on Reliability in Distributed Software and Database Systems*, 1986.
[20] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *EuroSys*, 2006.
[21] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the Future: A Framework for Automatic Malware Removal and System Repair. *ACSAC*, pages 163–176, December 2006.
[22] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. *FTCS-25*, pages 381–390, June 1995.
[23] B. Jack. Remote Windows Kernel Exploitation - Step into the Ring 0. *eEye Digital Security Whitepaper*, 2005.
[24] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: a safe dialect of C. *USENIX*, 2002.
[25] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and Efficiently Protecting the Heap. *ASPLOS*, October 2006.
[26] P. A. Lee and T. Anderson. *Fault Tolerance Principles and Practice 2nd. ed.* Springer-Verlag Wien New York, 1990.
[27] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protected Servers. *ACM CCS*, November 2005.
[28] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, pages 128–139, 2002.
[29] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. *ACM CCS*, November 2006.
[30] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005.
[31] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. *PLDI*, pages 158–168, June 2007.
[32] D. Oliveira, J. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. Chong. ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery. *ASID*, pages 381–390, October 2006.
[33] G. PortoKalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. *EuroSys*, April 2006.
[34] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. *ACM SOSP*, pages 235–248, October 2005.
[35] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. *MICRO-39*, pages 135–148, December 2006.
[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security through Failure-Oblivious Computing. *OSDI*, 2004.
[37] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processor. *ISCA-33*, 34(2):102–113, May 2006.
[38] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using Rescue Points to Navigate Software Recovery. *IEEE Symposium on Security & Privacy*, May 2007.
[39] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. *USENIX*, April 2005.
[40] A. Smirnov and T. cker Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. *NDSS*, February 2005.
[41] J. E. Smith and R. Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.
[42] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems. *International Symposium on Fault-Tolerant Computing*, 1991.
[43] F. Sultan, A. Bohra, P. Gallard, I. Neamtiu, S. Smaldone, Y. Pan, and L. Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, March/April 2005.
[44] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. *EuroSys*, March 2007.
[45] UnixBench. http://www.tux.org/pub/tux/benchmarks/System/unixbench/.
[46] Webstone 2.5. http://www.mindcraft.com/webstone/.
[47] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. *ACM CCS*, November 2005.
[48] bochs: the Open Source IA-32 Emulation Project (Home Page). http://bochs.sourceforge.net.
[49] Security Focus Vulnerability Notes, (http://www.securityfocus.com), bid == Bugtraq ID.
[50] The PAX team. http://pax.grsecurity.net.