Meaningful and Flexible Survivability Assessments: Approach and Practice©

Michael Atighetchi and Dr. Joseph Loyall
Raytheon BBN Technologies

With the increase of IT assets and attacks against them—along with increased attention to, and concern for, cybersecurity and the survivability of systems—systems assurance assessment has become more important than ever. This article describes a flexible process for assessing systems with respect to security and survivability. We discuss how our approach enables assessments throughout the project life cycle by tailoring the testing to a specific evaluation scope in terms of depth and coverage.

Current systems assurance testing tends to focus on functional and performance testing and often postpones security assessments until the end of the project life cycle—or after deployment or release. In our view, this shortcoming can be partially explained by the large cost and technical difficulty of formal security testing (e.g., during certification and accreditation), and the lack of standardized non-binary security metrics. For example, the testing guide of the Open Web Application Security Project (OWASP) [1] contains more than 340 pages on good security testing procedures but provides little guidance in selecting which tests to perform for a given application.

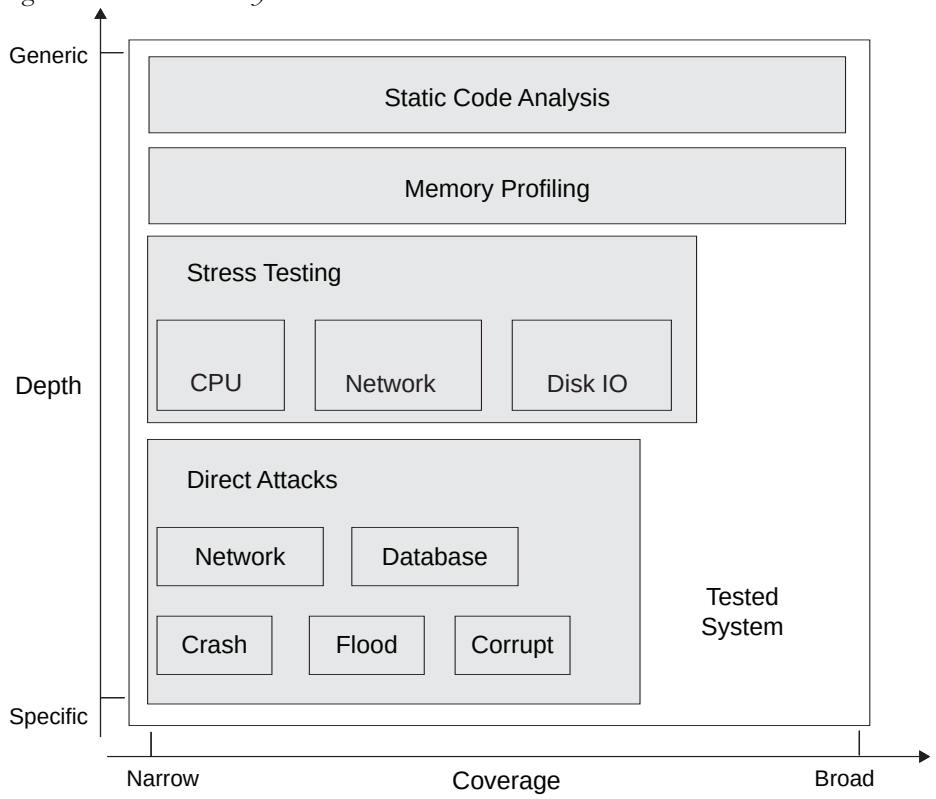
In this article, we present a flexible process for assessing evolving prototype systems with respect to security and survivability. Also discussed is how our approach enables assessments of confidentiality, integrity, and availability (CIA, a concept detailed in the sidebar) throughout the project life cycle by tailoring the testing to a specific evaluation scope in terms of depth and coverage. This process has enabled us to conduct security evaluations early in project life cycles and focus architecture and design efforts on addressing the shortcomings identified through repeated tests.

To show the testing methodology in action, this article provides details on a set of specific open-source and freely available tools we have found useful in our evaluations. The focus of this article is not on comparing different tools and their capabilities, but rather on showing, through insight from our *test attacks*, what vulnerabilities can be identified and how these tools can be reused across evaluations of different systems.

Finally, we report on the most recent assessment of an information management system (IMS), implementing a publish, subscribe, and query (PSQ) capability.

After giving a brief testing methodology overview, this article will detail static code analysis and memory profiling tools used in capturing a broad set of coding mistakes, present significant hands-on technical knowledge that can be used to study risk exposure CIA loss, and propose the next steps in both improving survivability assessments and mitigating vulnerabilities.

Figure 1: *The Survivability Evaluation Process*



Survivability Assessment Methodology

Continuous test and evaluation of security attributes of systems is an important part of testing as it allows software developers to identify and address vulnerabilities as part of the system architecture and design. This article describes a repeatable survivability testing methodology that optimizes the set of tests to execute given limited budget constraints and can run as part of continuous build processes. Figure 1 displays the main threads of our testing methodology as boxes.

First, we utilized existing tools for *static code analysis* and *memory profiling*. These tools provide excellent coverage at minimal cost but only support validation against a set of specialized known problems. Next, we performed a series of stress tests that simulated conditions found in operational environments, which tend to be less controllable than lab environments. Tests in this category focused on studying the impact of a large number of clients and large size information objects on critical server functionality. These tests were narrower in scope than static code checks and memory profiling, symbolized by the "Stress Testing" box in Figure 1. Finally, we developed a set of direct attacks against the system with different attacker privileges, ranging from attacks launched with only network layer

access to attacks that assumed corrupted clients. These attacks focused on exploiting specific vulnerabilities, therefore providing the least amount of coverage but the most amount of depth. Figure 1 visualizes our approach of subjecting the whole system to a set of low-cost generic checks to maximize coverage while performing a more in-depth analysis on carefully selected parts of the system. As the figure indicates, the methodology enables a flexible amount of testing from *specific* (i.e., to the requirements of the tested program) to *generic* (i.e., testing that could be applied to any program) and from *narrow* to *broad* coverage of the tested system. The depth and coverage of the testing can be chosen to enable the best testing possible in given time and budget constraints.

Static Code Analysis and Memory Profiling

The purpose of static code analysis is to automatically flag common coding errors that leave the system vulnerable to exploits. A large number of code analysis algorithms and tools for finding security vulnerabilities exist, including both commercial- and research-grade tools, with different trade-offs for cost and effectiveness.

We utilized several analysis tools as part of our research efforts (as described in the following text). We did not conduct a comprehensive comparison of the available analysis tools, but instead selected a representative set that provided useful analysis. There are alternatives to the particular tools that we selected, although the amount of coverage and analysis differs between tools. Furthermore, we concentrated on tools that analyze programs written in Java and C++. Arguably, these are the most common languages in use today and are the languages in which the systems that we tested were written. We believe the concepts and process that we put forward are valid for any language (although the set of available tools will certainly vary and the tools for languages other than Java and C++ might be less readily available).

FindBugs [2] was the most useful tool for us when analyzing Java code. FindBugs uses static analysis to inspect Java byte code for occurrences of bug patterns without the need to execute the program. For analyzing C++ code, we used FlawFinder [3] and RATS [4] open-source tools. FlawFinder is a program that examines source code and reports possible security weaknesses (*flaws*) sorted by risk level. It is useful for quickly finding and removing some potential security problems before a program is released. RATS is a tool for scanning C, C++, Perl, PHP, and Python source code, and for flagging common security-related programming errors such as buffer overflows and time-of-check-to-time-of-use race conditions.

In order to detect memory leaks (which can adversely affect availability) and memory corruption (which can be exploited to escalate privilege through buffer overflow attacks), this phase of the testing process runs a system through a set of memory profiling tools that keep track of dynamic memory allocations and perform analysis to detect errors during operation.

For C++ programs, we successfully used Valgrind [5], Mudflap [6], and mpatrol [7]. This set was selected to maximize coverage and mitigate errors introduced by individual tools. For Java programs, the scope of memory profiling is not to directly detect bad memory conditions (since Java avoids memory corruption and leak issues through garbage collection), but to provide useful debugging functionality in cases where Java runs out of memory.

Stress Testing

The goal of stress testing is to assess system functionality under application-level boundary conditions, such as high load scenarios with large information objects, large numbers of clients, and high rates of client requests.

Large MIOs

During the IMS assessment, we studied whether a single client can affect the availability of the IMS server by publishing a few very large managed information

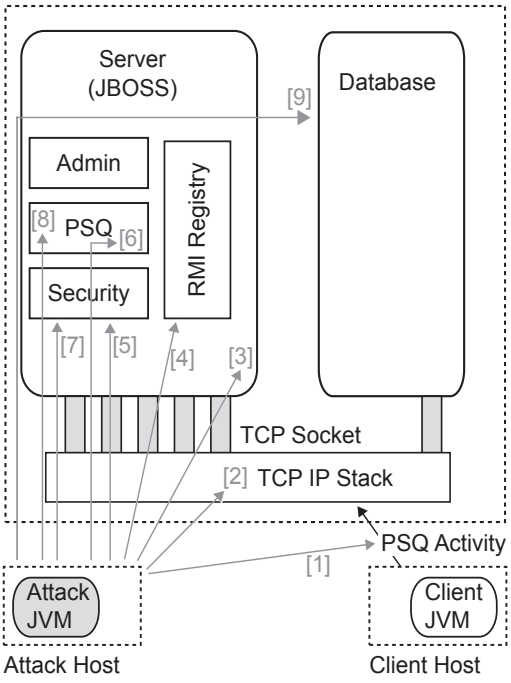
The CIA Triad

Confidentiality (C) is the assurance that information is not disclosed to unauthorized individuals, programs, or processes. For example, a credit card transaction on the Internet requires the credit card number to be transmitted from the buyer to the merchant and from the merchant to a transaction processing network. The system attempts to enforce confidentiality by encrypting the card number during transmission, by limiting the places where it might appear (e.g., in databases, backups, or printed receipts), and by restricting access to the places where it is stored. If an unauthorized party obtains the card number in any way, a breach of confidentiality has occurred.

Integrity (I) is the assurance that information is not altered in an unauthorized fashion. For example, integrity is violated when an employee deletes important data files, when a computer virus infects a computer, or when an employee is able to modify his own salary in a payroll database.

Availability (A) is the assurance that information, systems, and resources are available to users in a timely manner so productivity will not be affected.

Figure 2: *Progression of Direct Attacks that Compromise CIA*



[1] **Network Sniffing:** Break confidentiality.

[2] **TCP Connection Flooding:** Deny service to legitimate clients.

[3] **Serialization Attacks:** Take control over JVM.

[4] **MBean Backdoors:** Get unauthorized access to critical services.

[5] **Password Cracking:** Determine passwords through brute-force trial and error.

[6] **Malformed XML:** Crash PSQ logic.

[7] **SQL Injection:** Circumvent authentication and cause loss of confidentiality and integrity.

[8] **XPath/XQuery Injection:** Cause loss of confidentiality and integrity.

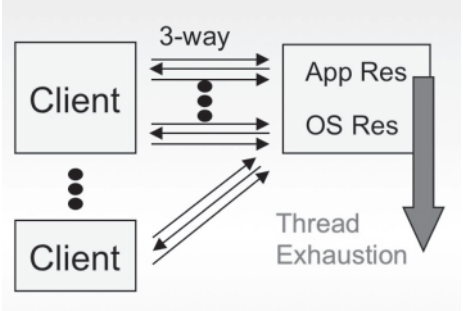
[9] **Simulated Node Failures:** Test liveliness monitoring.

objects (MIOs)[1]. To study the impact of MIO size, we modified an existing test client to publish MIOs with various metadata and payload sizes. The goal of this test was to identify the boundary point at which MIOs get rejected by the server due to their size. For scenarios with no active subscriptions, a client publishing an MIO with a combined payload size of 28MB (payload = 14MB and metadata = 14MB) caused exceptions in the core IMS and sometimes on the client side, leading to loss of the MIO. This was well below the maximum Java Virtual Machine (JVM) memory limits of 1024MB on both the client and server side and represented an inadvertent internal size limit that could be exploited by rogue clients. In a follow-on experiment, we studied the impact of the number of active subscriptions on the maximum accepted MIO size. One active subscription reduced the maximum MIO size to 5.8MB, while only MIOs with less than 2MB could be delivered to two subscribers. The expected cause was prolific copy operations on the metadata strings during subscription predicate matching. The IMS code has been tested with large payloads but never with MIOs with large metadata based on an undocumented assumption that metadata is usually small compared to payloads. While this assumption may hold in practice during normal use, it is the type of assumption likely to be exploited by a determined adversary. The solution to this problem is to refactor the code to avoid duplication of byte arrays.

Large Number of Clients

Another stress test focused on evaluating the impact of a large number of concurrent clients on IMS server availability. For that purpose, we simply started multiple publishing clients on the same client host. Out of memory exceptions occurred in the core IMS after registering 36 clients, resulting in a denial of service to all clients. In addition to identifying a point of optimization to increase the number of clients supported, this test also pointed out a survivability and security need. Since there is always going to be an upper limit to the number of clients that can be properly supported by a single server, there should be code in the server that checks the number of clients and refuses new clients when the limit is reached.

Figure 3: *TCP Connection Flooding Attack*



Direct Attacks

Adversaries typically don't start from scratch when attacking applications and can leverage a large collection of openly available tools to construct customized multi-step attacks targeting critical application functionality. Figure 2 (see previous page) displays a progression of attacks (with an increasing level of privilege) that an adversary might follow to compromise CIA. This attack sequence crosses multiple system and protection boundaries, as is common during sophisticated attacks. The following subsections describe each attack in more detail, describe observations made during test attacks, and provide suggestions on mitigation strategies.

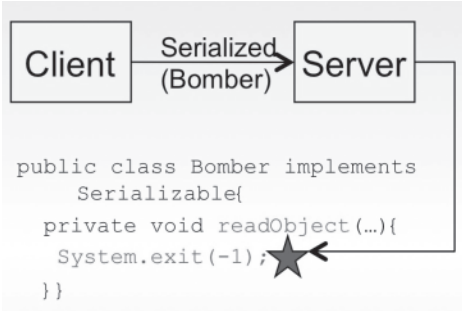
> *"Adversaries ... can leverage a large collection of openly available tools to construct customized multi-step attacks targeting critical application functionality."*

Network Sniffing

Description: Attackers use sniffing to find out information about the system (for attack purposes) and to steal sensitive information.

Risk: Loss of confidentiality and integrity.

Figure 4: *Java Serialization Attack*



Example: Wireshark [8] was used to capture the raw content of packets sent out by a publishing client to the IMS core. Although the IMS uses Transport Layer Security (TLS) [9] for encrypting data on the network and preventing sniffing attacks, it was noticed that TLS was only used for initial authentication and connection establishment between clients and the server. Published MIOs went over a separate Transmission Control Protocol (TCP) connection that did not provide TLS protection. As a result, the IMS was vulnerable to a loss of confidentiality via standard network sniffing. Furthermore, attackers could cause integrity violations by exchanging MIO payloads (e.g., swapping out targets in a target nomination list). Traffic was also vulnerable to replay attacks, which are a special form of data corruption where the data content isn't changed but corruption is still introduced by republishing MIOs.

Mitigation: The solution to these vulnerabilities is protecting all communication between clients and the server via TLS, and devising automated tests that classify all observed traffic.

TCP Connection Flooding

Description: The main idea of the TCP connection flood is to initiate and establish a large number of TCP connections from an infiltrated client to a listening socket. Since servers typically set aside memory and process resources for new connections, a large number of connections can create memory exhaustion in which further connections from legitimate clients will be dropped. The top of Figure 3 shows the normal *three-way handshake* during TCP connection establishment. A client initiates the connection via a synchronize (SYN) packet, which gets answered by the server with a SYN/Acknowledge (ACK) packet. Upon receiving the SYN/ACK packet, the client replies with another ACK packet, which establishes the TCP connection.

TCP connection floods are different from SYN floods, in which the client simply goes on to send out a large number of SYN packets originating either from the same client IP address or multiple (*spoofed*) client IP addresses without completing the three-way handshake. Each SYN packet needs to get processed by the server's TCP IP stack; such a situation can cause memory and CPU exhaustion in the server's TCP IP stack. Modern operating systems deal with SYN floods effectively using SYN cookies [10].

Figure 3 displays the sequence of events during a TCP connection flooding

attack. Here the client completes the three-way TCP handshake, but then keeps creating new connections. A straightforward version of this attack keeps the client's source IP of all connections the same, whereas more sophisticated versions can spread to multiple client IPs. Note, however, that this escalation cannot simply be achieved through IP address spoofing, since the client needs to get the server's SYN/ACK packet routed to it in order to proceed. Since each established connection generally causes application resources to be used (e.g., threads and associated memory structures), a large number of connections can cause applications to run out of memory, causing legitimate clients to fail.

Risk: Loss of availability.

Example: In an experiment to assess the IMS software's vulnerabilities to TCP connection floods, we used an attack client that established and held 1,200 TCP connections, and studied the effect of pointing this client to each of 15 listening ports in separate runs. The results were interesting. The maximum number of established connections varied across ports (from 60 to 753) as did attack effects. This included:

- A loss of publish functionality.
- A loss of administration functionality.
- A loss of system access on the node running the IMS server process due to maximum file handle limits.
- Denial-of-service through disk exhausting via large log files.
- No log generation for floods on certain ports, enabling attackers to execute stealthy attacks that are hard to diagnose.

Mitigation: A threshold scheme taking into account the source IP of the connection together with past connection history makes execution of this attack significantly harder. In addition, code to limit the rate of outgoing TCP connections from legitimate clients helps prevent such conditions in the case of accidental application programming interface misuse in client programs. Furthermore, ports that only require local access (e.g., 5400) should be bound to 127.0.0.1 instead of 0.0.0.0 to prevent remote execution of connection flooding attacks. Finally, all code should be augmented with proper exception handling to generate a small number of succinct exceptions in flooding conditions.

Serialization Attacks

Description: This attack exploits known type safety issues Java programs encounter during deserialization. Creating and sending a special serialized Java object to a remote method invocation (RMI) server port (as shown in Figure 4) allows attackers to remotely exploit JVM bugs and cause client-initiated execution of arbitrary code.

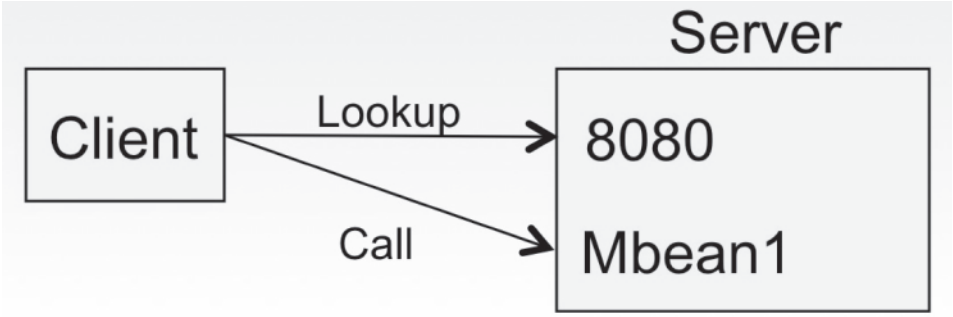


Figure 5: *Direct Calls on MBeans Without Authentication*

Risk: Loss of CIA.

Example: Let's consider the following server code snippet used for reading objects off the network, as described in [11]:

```
  mySocket = new ServerSocket(3000);
  while (true) {
    Socket client = mySocket.accept();
    ReceiveRequest dtwt = new Receive
Request (client);
}
class Request implements Serializable { }
class ReceiveRequest extends Thread{
    Socket clientSocket = null ;
    ObjectInputStream ois = null;
    ReceiveRequest (Socket theClient)
throws Exception {
     clientSocket = theClient;
     // get the Streams
     ois = new
          ObjectInputStream(clientSocket.
get InputStream());
}

   public void run() {
    try {
      Request ac = (Request)
                          t=2
                    ois.readObject(); }
                          t=1
     catch (Exception e)
{  System.out.println(e) ; }
      // ...
    }
 }
```

Note how the server first reads in the Object (t=1) and then performs the cast to the expected Request type (t=2). The generated byte code shows the following execution sequence:

```
public void run();
     Code:
     0:   aload_0
t=1  1:   getfield      #3; //Field
                        ois:Ljava/io/Object
                        InputStream;
     4:   invokevirtual  #7; //Method
                        java/io/ObjectInputStream.
                        readObject:()
                        Ljava/lang/Object;
t=2  7:   checkcast     #8; //class Request
    10:   astore_1
    11:   goto     22
    14:   astore_1
    ...
```

The sequence of events is as follows:

- t=0 client sends byte stream (serialized object data) via ObjectInputStream.
- t=1 Server branches into the readObject method of the class according to the result returned by getfield.
- t=2 server casts the object to the needed type.
  - Cast is valid: continue work.
  - Cast is invalid: throw ClassCast Exception.

Between t=0 and t=2, there is no type safety. The client gets to decide (at t=0) which code the server branches into (at t=1). This opens up an attack path in conjunction with an existing vulnerability of a readObject method on any class that is on the server's load path. First, attackers can find some vulnerable class definitions on the server (especially in a readObject method, any serializable class will do). Next, they can construct an object according to this class definition and finally embed the malicious object in the ObjectInputStream payload of the Java 2 Platform Enterprise Edition (J2EE) protocol (RMI, RMI/Internet Inter-Orb Protocol, Java Naming and Directory Interface [JNDI], and so forth). Serialization attacks have been used frequently in the past: see [11] for regular expression exploits and [12] for exploiting hash table collisions.

Mitigation: Our suggestions to prevent serialization vulnerabilities involve careful review and minimization of loadable classes on the server's classpath. In addi-

tion, a crumple zone—a layer of proxy components that anyone seeking service must go through—can be deployed to perform the deserialization of all objects entering the core. Augmenting proxies with proper monitoring and restart capabilities, and introducing diversity through different JVM implementations and programming languages, will help reduce the dangers of this attack type.

MBean Backdoors

Description: This attack attaches a remote client to the managed beans (MBeans) available—via Java Management Extensions inside of a J2EE server—and makes dynamic calls into the server without the need for authentication. As shown in Figure 5 (see previous page), clients can make direct calls on MBeans handled by a JBoss Application Server (AS) by simply performing a lookup, creating an RMIAdaptor connection, and making RMI calls.

Risk: Loss of CIA.

Example: In the IMS code, clients could directly connect to the MBeans via jmx/invoker/RMIAdaptor and make calls through the MBeanServerConnection without needing to authenticate. This would enable calls that could remove all repositories, inject a man-in-the-middle master repository (loss of confidentiality), and change security policies (loss of integrity), for example. No logging is performed during these remote calls, which makes this attack very stealthy.

Mitigation: To prevent this attack, one needs to harden access to 127.0.0.1:8080/invoker/JNDIFactory and lookup of jvm/invoker/RMIAdaptor. Using TLS and binding, the listening socket to 127.0.0.1 only will make access by unauthorized clients more difficult. In addition, tighter integration of the socket listener on port 8080 with security handling code helps prevent this attack.

Password Cracking

Description: This attack guesses correct passwords through repetitive trials of passwords retrieved from common attack dictionaries. Once the valid administrator password has been determined, the adversary can simply log in as the administrator and perform all actions normally granted to administrators, including deletion of critical MIOs and removal of users.

Risk: Loss of CIA.

Example: The prototype IMS code was susceptible to brute force password attacks because it allowed attack scripts to try an unlimited number of password combinations without impacting account status. No log events occurred in the JBoss AS console when authentication failed, making the attack difficult to detect.

Mitigation: An effective approach to counter-password attacks is to establish a cutoff scheme in which accounts are locked down after a specific number of failed login attempts. In addition, failed login attempts should be logged to a management station.

Malformed XML

Description: These attacks attempt to deny service to legitimate clients by taking over a single client and publishing malformed XML input data with the intent to crash the server.

Risk: Loss of availability.

Example: We tested the vulnerability of the prototype IMS to this kind of attack by modifying the client to publish MIOs with improperly formatted metadata. XML validation properly caught and handled the bad content. Next, we tried uploading a malformed schema and observed the following issues:

- Schemas were properly validated, but validation exceptions cause ClassNotFoundExceptions.
- Schemas that don't start with <?xml version passed validation but caused errors during publication.
- Schemas with duplicate xsd:schema lines passed validation but cause Null Pointer Exceptions in the Berkeley DB XML backend, and cause Web console access to the Interoperable Object Reference to fail.

These issues are easy to fix but are indicative of the bugs that can creep in, even when XML content is validated through schemas but the schema itself is not.

Mitigation: The use of restrictive XML schemas and validation of all XML input against them goes a long way in addressing this vulnerability.

SQL Injection

Description: SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and, thereby, unexpectedly executed. It is (in fact) an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.

Risk: Loss of CIA.

Example: Figure 6 displays the normal flow of data from a client via a server to the backend SQL DB. If the data sent by the client is not filtered before reaching the SQL DB, the client can execute arbitrary commands on it.

Figure 6 depicts one such example that was used by attackers during the 2005 OASIS Dem/Val red team exercise [13]. The attack locked up servers by inserting a BENCHMARK command, which would cause the SQL DB to use 100 percent of the available CPUs. The BENCHMARK command causes MySQL (a relational database management system) to run through an extensive set of mathematically expensive computations, pre-empting useful computation for hours.

Our testing indicated that the prototype IMS code was not vulnerable to SQL injection attacks.

Mitigation: To prevent SQL injection attacks, user interfaces should be designed to be as restrictive as possible (e.g., a selection menu instead of free text entry, and

Figure 6: *Data Propagation to SQL Databases (SQL DBs)*

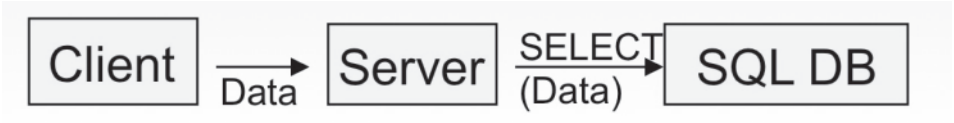
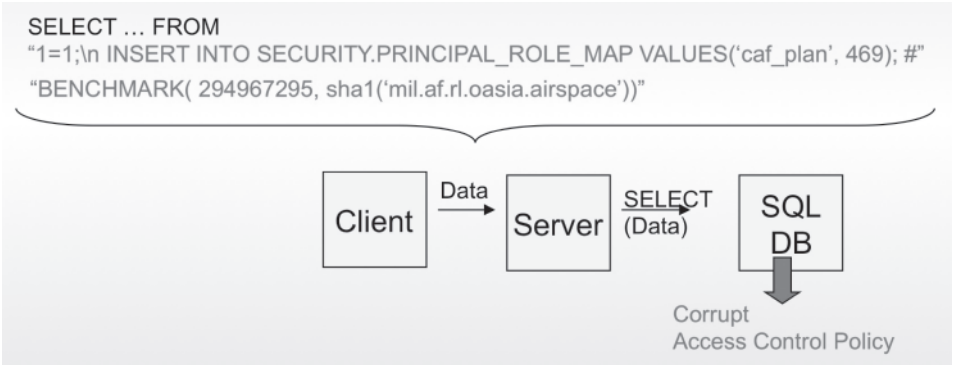


Figure 7: *Example SQL Injection Attack*



all user input should be checked for embedded SQL commands).

XPath/XQuery Injection

Description: This attack attempts to gain the same effects described in the SQL injection attack by providing specially crafted query predicates.
Risk: Loss of CIA.
Example: The following code fragment shows an example in which the client explicitly specifies parts of the XQuery that usually gets implicitly generated and executed by the IMS server:

```
connection.createQuerySequence
        ("mil.af.rl.oim.training.xmlx-
path", "1.0");
String xmlxpathPredicate =
Utils.createPredicate
  ("TestPredicate", "XQuery",
   "for $a in
collection(\"SCHEMA_422547673.dbxml\")
     /child::sys:Metadata
     where ($a//ATO/type = \"ATO\")

returndbxml:metadata(\"dbxml:name\",
$a)");
```

XPath injection attacks proved partially effective against the prototype IMS, partly because it dynamically constructed and executed XQuery statements from XPath input. Custom XQuery predicates allow a small amount of information exfiltration. For instance, an attack client could probe through repeated queries to determine whether air task orders (ATOs) were available with certain metadata fields, although the client could not actually retrieve the ATO payload. In addition, the IMS, which explicitly prohibited full XQuery predicates because of their ability to modify the database, included a *backdoor* through which full XQuery commands could be sent to the database. This only presented a vulnerability if the backdoor code persisted through releases of the code. In addition to vulnerabilities to the IMS database, XQuery is a powerful language, allowing (in principle) direct calls into the JVM through external functions. This would allow an attacker, for instance, to call `System.exit(-1)` through a specially crafted query. The prototype IMS did not exhibit this vulnerability because it used an earlier version of XQuery that did not support embedded Java functions.
Mitigation: To prevent XQuery exploits, all user input should be checked for dangerous XQuery commands, such as `delete`, `update`, and references to external functions.

Software Defense Application

This article describes survivability assessment techniques that will benefit the defense software community through more flexible design evaluations, more useful assessments, and reduced time in identifying and mitigating vulnerabilities. The in-depth analysis of the most prevalent cyberattack scenarios will help defense software developers understand what they may encounter; but, more importantly, the authors present first-hand accounts detailing how they solved these problems. The end result is a significant increase in a system's overall security.

Simulated Node Failures

Description: A common survivability experiment involves studying the impact of crashes of one component on other components. However, it can be difficult to devise an attack that is so precise that it crashes only the targeted component. Therefore, this attack uses fault injection techniques to simply cause the desired fault (e.g., by manually stopping a component).
Risk: Sustained loss of availability.
Example: We injected crash faults in the IMS database (Berkeley DB XML) by sending the database a `kill -9` signal. The IMS did not contain a monitoring protocol to test the liveliness of processes. An accidental crash of the Berkeley DB XML process led to situations in which the IMS server was unavailable, and the situation was noticed only when critical operations started failing.
Mitigation: A secure, heartbeat-based monitor that provides low failure detection latencies and is resistant to spoofing attacks. The monitor can reduce the impact and speed the recovery from these attacks.

Results and Summary

Table 1 displays a summary of both stress tests and direct attacks conducted, their results, and their effects on CIA.

In summary, nine of the 11 tests were successful in achieving attack objectives. Risks to confidentiality were exposed in five tests, which is a surprisingly high number given that it is generally much easier to cause a denial of service than to exfiltrate data without being detected. The experiment generated new requirements for the IMS—both in terms of fixing bugs and ease-of-use issues as well as in addressing deeper problems, including a single point of failure for various components, lack of adequate management tools, and susceptibility to direct attacks

This article presented a flexible process for evaluating systems early on in their life cycles for information survivability and showed its application during an assessment of a prototype PSQ information management system. This work

Table 1: *IMS Assessment Results*

Test	Result	Effect*
Large MIOs	Out of memory errors with IOs over md = 14MB, pl = 14MB, 0 subscribers md = 5.8MB, pl = 5.8MB, 1 subscriber md = 2MB, pl = 2MB, 2 subscribers.	A
Large Number of Clients	Out of memory errors after registration of 36 clients.	A
Network Sniffing	MIOs are sent in the clear.	CI
TCP Connection Flooding	Denial-of-service without generating log entries.	A
Serialization Attacks	Clients can execute arbitrary code in the JVM without authentication.	CIA
MBean Backdoors	Clients can make anonymous calls on MBeans.	CIA
Password Cracking	Brute-force password testing at the rate of 100 per second.	CI
Malformed User Data	Apollo resilient against malformed metadata and schemas.	None
SQL Injection	Suspicious SQL code turns out to be dead code.	None
XQuery Injection	Unauthorized access to MIOs and execution of arbitrary code.	CIA
Simulated Node Failures	Absence of monitoring protocol.	A

*C = Confidentiality, I = Integrity, A = Availability pl = Payload, md = Metadata

builds upon previous and ongoing research in survivable distributed systems and addresses the current and future need to effectively and accurately evaluate system guarantees in the presence of sophisticated cyberattacks. We have packaged a number of reusable attack techniques and associated tools that can be customized for new systems with little overhead.

Next Steps

Our future research will continue to build upon the work presented in this article in two ways.

First, we plan to extend our assessment approach by adding more techniques and automating customization and automation aspects of attack scenarios. This will further reduce overhead costs for assessments and increase their frequency during a project life cycle. We clearly see the value of establishing a community-based collaboration platform for survivability assessments (e.g., those hosted on <https://www.forge.mil>). As more projects start using the techniques described in this article, system survivability requirements will need to be taken into account as systems need to provide different levels of survivability. More research is needed to develop a point/scoring system that evaluates survivability in a systematic and quantitative way—given the security posture and criticality of an application.

Second, we plan on investigating solutions to the deeper problems of the existence of single points of failures and the lack of adequate management tools. For future work in service-oriented information management systems, we intend to:

1. Extend service-oriented architecture and design techniques to include security and survivability concepts that facilitate survivable designs.
2. Develop security services, mechanisms, and execution containers for preserving system CIA in the presence of cyberattacks.
3. Develop an environment to assess and evaluate composition patterns, enabling customization of tradeoffs between survivability, performance, and functionality for specific environments.◆

Acknowledgements

The authors would like to acknowledge the support and collaboration of the U.S. Air Force Research Laboratory (AFRL) Information Directorate and the ITT Corporation. This work was sponsored by the AFRL under contract number SPO700-98-D-4000, Subcontract Number: 205344 – Modification No. 5.

About the Authors

Michael Atighetchi is a scientist at Raytheon BBN's Information and Knowledge Technologies business unit. His research interests include cross-domain information sharing, security and survivability architectures, and middleware technologies. Atighetchi has published more than 35 technical papers in peer-reviewed journals and for conferences, and is a senior member of the IEEE. He holds a master's degree in computer science from the University of Massachusetts at Amherst, and a master's degree in IT from the University of Stuttgart, Germany.

Raytheon BBN Technologies
10 Moulton ST
Cambridge, MA 02138
Phone: (617) 873-1679
E-mail: matighet@bbn.com

Joseph Loyall, Ph.D., is a principal scientist at Raytheon BBN Technologies. He was the principal investigator for Defense Advanced Research Projects Agency and AFRL research and development projects in the areas of information management, distributed middleware, adaptive applications, and quality of service. He is the author of more than 75 published papers; was the program committee co-chair for the Distributed Objects and Applications conference (2002, 2005); and has been invited speaker at several conferences and workshops. Loyall has a doctorate in computer science from the University of Illinois.

Raytheon BBN Technologies
10 Moulton ST
Cambridge, MA 02138
Phone: (617) 873-4679
E-mail: jloyall@bbn.com

References

1. OWASP Foundation. *OWASP Testing Guide*. Vers. 3.0. 2008 <www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf>.
2. The University of Maryland. "Find Bugs – Find Bugs in Java Programs." *SourceForge.net*. 21 Aug. 2009 <http://findbugs.sourceforge.net>.
3. Wheeler, David W. "FlawFinder." <www.dwheeler.com/flawfinder>.
4. Fortify Software, Inc. "Welcome to RATS – Rough Auditing Tool For Security." 2009 <www.fortifysoftware.com/security-resources/rats.jsp>.
5. Valgrind Developers. "Valgrind." <http://valgrind.org>.
6. GCC, the GNU Compiler Collection Wiki. "Mudflap Pointer Debugging." 10 Jan. 2008 <http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging>.
7. Roy, Graeme S. "mpatrol." 16 June 2009 <http://mpatrol.sourceforge.net>.
8. Wireshark Foundation. "About Wireshark." <www.wireshark.org/about.html>.
9. Wagner, David, and Bruce Schneier. *Analysis of the SSL 3.0 Protocol*. Proc. of the Second USENIX Workshop on Electronic Commerce. Oakland: 18-20 Nov. 1996. Paper Revised 15 Apr. 1997 <www.schneier.com/paper-ssl-revised.pdf>.
10. Bernstein, D.J. "SYN Cookies." <http://cr.yp.to/syncookies.html>.
11. Schönefeld, Marc. *Pentesting J2EE*. Proc. of Black Hat Federal 2006. 25 Jan. 2006 <www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>.
12. Crosby, Scott A., and Dan S. Wallach. *Denial of Service via Algorithmic Complexity Attacks*. Proc. of the 12th USENIX Security Symposium. Washington, D.C.: 4-8 Aug. 2003 <www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf>.
13. Pal, Partha, Franklin Webber, and Richard Schantz. *The DPASA Survivable JBI—A High-Water Mark in Intrusion-Tolerant Systems*. Proc. of the 2nd EuroSys Workshop on Recent Advances in Intrusion-Tolerant Systems. Lisbon, Portugal: 23 Mar. 2007 <http://wraits07.di.fc.ul.pt/4. pdf>.

Note

1. A unit of information consisting of payload (the information) and metadata describing the information and used for brokering.