Static and dynamic testing in the software development life cycle

Three approaches to improving the security of your software products

M. Tim Jones

Yesterday, the idea of application security was mostly an afterthought. But given the plethora of news on hacking and underground economies for exploits, security testing is now an integral part of the software development life cycle. This article explores two aspects of security testing and the open source tools that simplify their execution.

In the past decade, the art and practice of hacking has taken a significant turn for the worse. From the volume and complexity of attacks to the growing audience of international participants who hack for fame or fortune, hackers are modern-day pirates seeking adventure on the high seas of the Internet.

But what makes this trend even more critical is the size of the attack surface the Internet makes possible. We live in an increasingly connected world, where physical or package security is no longer the hacker's obstacle. Instead, knowledge of network protocols, applications, and an ever-growing list of exploits and utilities make up the hacker's toolkit.

Spectrum of hacking

IBM Security AppScan® is a leading application security testing suite designed to help manage vulnerability testing throughout the software development life cycle. IBM Security AppScan automates vulnerability assessments and scans and tests for all common web application vulnerabilities, including SQL injection, cross-site scripting, buffer overflow, and new flash/flex application and Web 2.0 exposure scans.
Today, hacking is big business. Previously unknown exploits in software, called **zero-day attacks**, can be quite profitable. For example, selling zero days in the underground markets can yield $250,000 USD (for Apple iOS exploits and less as a function of the popularity or robustness of the software).

At the bottom end of the spectrum are novice hackers. Hacking is not only practiced by those who study the art but also by those who lack skills. So-called **script kiddies** are able to use the tools of the trade the hacking community (in addition to white hat penetration testers) have developed to wreak havoc on legacy applications or those with known exploits. At the top end of the spectrum are corporate, government, and private groups and individuals who hack for information, control, money, or brief fame. These individuals have the expertise and backing that make them the most dangerous not only to our deployed applications but also to entire industries (financial, energy, utilities, and more).

Regardless of the reason, the result is a near-constant siege on the software that we write every day. As developers, our jobs were once writing software that simply worked. Today, our software must not only work but do so reliably under constant threat.

**Test approaches**

Although security was once an afterthought in software development, it must now be considered throughout the entire software development life cycle (SDLC). For this article, I divide the life cycle into three separate phases: design, development and verification, and production. The design phase is a wetware process in which you identify requirements and produce an architecture and design. During development and verification, you write and test your system. Finally, in production, you deploy and maintain your system.

Securing your system requires different approaches and tools as a function of your phase in the life cycle (see Figure 1). During the design phase, you rely on good, secure design processes and reviews (and possibly some formal methods such as specification or modeling languages). In the development and verification phase, you have code that you can touch and test as well as perfect for automated review and inspection while under execution. In production, you can inspect the application under execution. Automated reviews and inspecting applications under execution go by special names—**static analysis** and **dynamic analysis**, respectively.
Static and dynamic testing in the software development life cycle

Figure 1. Static and dynamic testing in the SDLC

Static analysis provides several techniques for analyzing source (or object) code not under execution. In this category, there are tools such as Secure Programming Lint (Splint) and IBM® Security AppScan® Source. Dynamic analysis tools rely on executable code to analyze or instrument to extract information. In the dynamic category there are tools such as Valgrind and IBM Security AppScan Standard and Enterprise.

There is also another form of dynamic testing called network reconnaissance. Tools in this subcategory can help identify vulnerabilities in an application through remote network access.

Tools for static testing

Static analysis is the examination of source code (or object code after compilation). Using a variety of methods such as data-flow analysis, static analysis tools can uncover issues such as memory leaks, buffer overflows, and even concurrency issues. Static analysis works by scanning one or more source files and creating a representation of the scanned source to analyze it (see Figure 2).

Figure 2. Generic architecture of a static analysis tool

One of the most interesting open source tools from this category is a program called Splint. Lint was a UNIX® utility that first appeared in 1979 that could be used to flag suspicious or nonportable constructs within a program. Splint has now adopted the focus of security vulnerabilities, given the rising emphasis on this issue, based on the idea from Lint.
Listing 1 provides a simple example of Splint. As shown, the simple (poorly written) function should get the PATH environment variable and write it into a passed string argument. Splint finds several issues with this (some pedantic items were removed for brevity). In particular, there’s no guarantee that the passed string is not NULL and that sufficient space exists in the passed string to hold the contents of the getenv operation. Both are valid issues, and the second makes the program insecure for buffer overrun attacks.

**Listing 1. Demonstrating static analysis with Splint**

```
$ cat getpath.c
void getPath( char *str )
{
    strcpy(str, getenv("PATH"));
}

$ splint getpath.c -strict
Splint 3.1.2 --- 03 May 2009

gopath.c: (in function getPath)
gopath.c:3:3: Undocumented modification of *str possible from call to strcpy:
gopath.c:3:15: Possibly null storage passed as non-null param:
getpath.c:3:3: Possible out-of-bounds store: strcpy(str, getenv("PATH"))
   A memory write may write to an address beyond the allocated buffer. (Use -boundswrite to inhibit warning)
```

One thing you will notice in Listing 1 is Splint recommending "control comments." These are annotations that can be applied to the source inside comments, which allows a developer to further refine information to Splint about the program for more detailed analysis.

Splint is not the only utility that provides static analysis through source scanning. Although you can use Splint to analyze C and C++ programs, other utilities focus on other languages and application frameworks. Table 1 provides a short list of other static analysis tools and their target language or framework.

**Table 1. Sampling of static analysis tools**

<table>
<thead>
<tr>
<th>Language or framework</th>
<th>Static tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>C or C++</td>
<td>Splint, VisualCodeGrepper</td>
</tr>
<tr>
<td>Java™ technology</td>
<td>FindBugs, LAPSE+, VisualCodeGrepper</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>JavaScript</td>
<td>JSLint, JSHint</td>
</tr>
<tr>
<td>Python</td>
<td>pylint, PyChecker</td>
</tr>
<tr>
<td>PHP</td>
<td>RIPS</td>
</tr>
<tr>
<td>Ruby on Rails</td>
<td>Brakeman, codesake_dawn</td>
</tr>
</tbody>
</table>

From a professional perspective, there's also the commercial IBM Security AppScan Source. This product provides a complete portfolio solution for traditional and mobile applications as well as full-coverage insight of threats for audit and compliance purposes. In addition to the technical insight that it provides, it is able to aggregate with dynamic tests for a hybrid perspective of security risks and includes a variety of dashboards for high-level statusing. Covering a variety of popular languages (such as C/C++, PHP, SAP, Java, and Microsoft® .NET) and integration with integrated development environments, Security AppScan Source is ideal for both security testing and risk management.

Finally, don't forget about your compiler toolchain when exploring protection for security. Most compilers include enhanced error checking that is typically disabled by default, but once enabled, it can help to uncover significant issues within your source.

**Tools for dynamic testing**

*Dynamic analysis* is the examination of a program during run time. Like static analysis, dynamic analysis uses a number of techniques as a function of the data to be extracted. You can use dynamic analysis to identify code coverage (or the paths taken in a given application). This is useful, because paths not taken in an application are likely untested and contain bugs or exploits. Dynamic analysis works by either integrating introspective code into an application at build time (the most common approach) or providing a form of platform emulation to understand the internal behavior of an application during execution (see Figure 3).
An interesting example of dynamic analysis is the Valgrind tool. Valgrind gets its introspection capabilities by running between an application and the physical platform. The application executable is translated into an intermediate representation with instrumentation applied. This intermediate representation is then translated into the physical machine code using just-in-time (JIT) compilation techniques to provide the best performance possible. Valgrind provides several tools to analyze a program that's under execution. These tools can check for uninitialized memory, invalid use of dynamic memory, leaks of dynamic memory, heap profiling, detecting race conditions in multithreaded code, and more.

Let's explore a simple example with Valgrind. This example shows a simple erroneous program to see what Valgrind detects and how it reports the error.

The sample application, build, and test are shown in Listing 2. Notice that there are two issues in the badmem function. The first is an attempt to write beyond the bounds of a buffer, and the next is a buffer allocated but never freed.

**Listing 2. Demonstrating dynamic analysis with Valgrind (sample application)**

```
// mem.c
#include <stdlib.h>

char* badmem( void )
{
    char *ptr = malloc( 8 );
    ptr[8] = 0;
    return ptr;
}

void main( void )
{
```

Figure 3. Generic architecture of a dynamic analysis tool (introspective and platform)
char *result;
result = badmem();
return;
}

Next, I prepare my program for Valgrind and then execute it within the Valgrind virtual machine (VM; see Listing 3). I build my sample application with optimizations disabled (-O0) and also with debug information (-g) so that I can receive symbols in my output (function names, line numbers). When built, I execute Valgrind, enabling leak checking and specifying my test application. Note in the output that Valgrind detected my out-of-bounds write, and as part of the heap summary, I can see that 8 bytes were allocated from the heap and were still in use.

Listing 3. Demonstrating dynamic analysis with Valgrind (sample run)

$ gcc -g -O0 mem.c -o mem
$ valgrind --leak-check=yes ./mem

==17340== Memcheck, a memory error detector
==17340== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==17340== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==17340== Command: ./mem
==17340== Invalid write of size 1
==17340==    at 0x80483FF: badmem (mem.c:7)
==17340==    by 0x8048414: main (mem.c:16)
==17340== Address 0x419b030 is 0 bytes after a block of size 8 alloc'd
==17340==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==17340==    by 0x80483F5: badmem (mem.c:5)
==17340==    by 0x8048414: main (mem.c:16)
==17340==
==17340== HEAP SUMMARY:
==17340==     in use at exit: 8 bytes in 1 blocks
==17340== total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==17340==
==17340== LEAK SUMMARY:
==17340==     definitely lost: 0 bytes in 0 blocks
==17340==     indirectly lost: 0 bytes in 0 blocks
==17340==      possibly lost: 0 bytes in 0 blocks
==17340==     still reachable: 8 bytes in 1 blocks
==17340==     suppressed: 0 bytes in 0 blocks
==17340== Reachable blocks (those to which a pointer was found) are not shown.
==17340== To see them, rerun with: --leak-check=full --show-reachable=yes
==17340==
==17340== For counts of detected and suppressed errors, rerun with: -v
==17340== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 7)
$

For larger programs, you'll notice some degradation in performance. Because Valgrind is dynamically translating (from the binary to an intermediate representation, and then back to execute on the underlying architecture), some loss of performance does occur.

Valgrind is one example of a dynamic analysis tool (see Table 2). A similar tool called DynamoRIO supports the run time code manipulation for program analysis (enabling profiling, instrumentation, optimization, and translation). Another example of this class is Pin, which allows modes of operation for JIT compilation and a probe mode that minimizes the overhead of a running program at the cost of a limited feature set.
Outside of the VM or emulation approaches, introspective tools for dynamic analysis are also available. One of the most common is GNU’s gcov and gprof, which provide coverage analysis and profiling using compile-time instrumentation. The Debug Malloc (dmalloc) library provides drop-in replacements for Malloc/free (and its variants) to enable run time tracking of resources for memory leaks, out-of-bounds writes, and other capabilities.

Table 2. Sampling of dynamic analysis tools

<table>
<thead>
<tr>
<th>Dynamic analysis style</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic translation/VM</td>
<td>Valgrind, DynamoRIO, Pin</td>
</tr>
<tr>
<td>Compile-time instrumentation</td>
<td>gcov, gprof, dmalloc</td>
</tr>
</tbody>
</table>

Finally, one potentially underused capability to protect against anomalies is the manual assert. Although not ideal in production code, the assert function (which causes a program to exit upon detecting a catastrophic or unrecoverable issue) is useful for identifying bugs during development. Most languages implement a form or assertions and can help to pinpoint issues that could be the next exploit opportunity.

Tools for vulnerability testing (dynamic)

The last class of dynamic testing explored is vulnerability scanning. This form of testing permits what is called network reconnaissance and is popularly known as penetration testing. For web-based applications that are Internet facing, this type of analysis is key to ensuring a robust and secure application. You can perform this testing prior to application release and also when the application is deployed and in operation.

Vulnerability scanning involves a program designed to access devices in search of weaknesses. The touchpoints of a vulnerability scanner are devices, ports, and protocols exposed through those ports. Figure 4 provides the generic architecture of a vulnerability scanner.

Figure 4. Generic architecture of a vulnerability scanning tool
The most popular port scanner is a program called Network Mapper (nmap). The nmap utility is quite broad and extensive and includes a scripting engine that allows you to write scripts to interact with a target (using the Lua scripting language). Listing 4 provides a demonstration of nmap in a simple example. I start with a simple port scan of the entire network (using -sP, or Ping Scan). This results in a list of targets with their names and IP addresses. Of interest is an HP printer. Next, I point nmap at the printer and request a port scan (and assume that the host is up using -PN). This provides me with a list of ports through which I can access the device in an attempt to take control of the device. I then dig a little further to see what’s behind the HTTP port (80) using Telnet to retrieve more information (printer name, serial number, ID of the application-specific integrated circuit [ASIC], and so on). The next step is to search for available exploits of those ports for an HP printer and apply an exploit framework such as metasploit to take advantage of them.

Listing 4. Demonstrating the nmap utility for network reconnaissance

```
$ sudo nmap -sP 192.168.1.*
Starting Nmap 5.00 ( http://nmap.org ) at 2013-07-29 11:26 MDT
Host Frylock (192.168.1.1) is up (0.0018s latency).
...
Host mtjs-iPhone.hsd1.**********.net (192.168.1.100) is up (0.0011s latency).
Host 192.168.1.101 is up (0.0011s latency).
Host Megans-iMac.hsd1.**********.net (192.168.1.108) is up (0.0027s latency).
Host marcs-iPod.hsd1.**********.net (192.168.1.123) is up (0.00030s latency).
Host HPA20BE7.hsd1.**********.net (192.168.1.125) is up (0.00035s latency).
...
$ sudo nmap -PN 192.168.1.125
Starting Nmap 5.00 ( http://nmap.org ) at 2013-07-29 11:31 MDT
Interesting ports on HPA20BE7.hsd1.**********.net (192.168.1.125):
Not shown: 987 filtered ports
PORT     STATE SERVICE
80/tcp   open  http
139/tcp  open  netbios-ssn
443/tcp  open  https
445/tcp  open  microsoft-ds
515/tcp  open  printer
631/tcp  open  ipp
743/tcp open  unknown
8080/tcp open  http-proxy
9100/tcp open  jetdirect
9101/tcp open  jetdirect
9110/tcp open  unknown
9111/tcp open  DragonIDSConsole
9220/tcp open  unknown
Nmap done: 1 IP address (1 host up) scanned in 89.84 seconds
$
$ telnet 192.168.1.125 80
Trying 192.168.1.125...
Connected to 192.168.1.125.
Escape character is '^]'.
HEAD / HTTP/1.1
HTTP/1.1 200 OK
Server: HP HTTP Server; HP Officejet Pro 8600 - CM749A; Serial Number: CN3CMCZGG505KC;
Coulomb_base_pp Built:Fri May 31, 2013 08:21:00PM {CLV1CM1342CR, ASIC id 0x00320104}
Last-Modified: Fri, 31 May 2013 20:21:06 GMT
...
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```
The point of this demonstration is that when building network or Internet-facing devices, you must be diligent about what you expose. Every open port is a potential vulnerability, and the fewer ports (and information exposed through those ports), the more secure your device will be.

Many tools exist in the category of vulnerability testing, some related to port and network scanning, others for targeted vulnerability scanners, and still other tools for fuzzing and low-level analysis. You'll find a list of these tools in Table 3. From this table, port scanners enumerate the available ports in a network or on a host and implement a form of application fingerprinting to find out what's there (such as the host's operating system or an application behind a port). Vulnerability scanners provide a framework through which ports can be exploited, using a catalog of known exploits on a per-operating system, application, and version basis. Vulnerability scanners are now segmented based on applications (one example is the database class of security scanner). Web scanners are similar to vulnerability scanners but focus solely on web systems. Fuzzing tools provide the means to generate random (invalid or unexpected) input to an application to test the boundaries of its error checking. Finally, low-level network tools (such as Wireshark) provide the means to inspect the details of protocol traffic.

Table 3. Sampling of network reconnaissance tools

<table>
<thead>
<tr>
<th>Network reconnaissance style</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port scanners</td>
<td>nmap, scanrand</td>
</tr>
<tr>
<td>Vulnerability scanners</td>
<td>Metasploit, OpenVAS</td>
</tr>
<tr>
<td>Web scanners</td>
<td>Nikto, sqlmap</td>
</tr>
<tr>
<td>Fuzzing tools</td>
<td>w3af, Skipfish, Wfuzz</td>
</tr>
<tr>
<td>Low-level network tools</td>
<td>tcpdump, Wireshark</td>
</tr>
</tbody>
</table>

From a professional perspective, there's also Security AppScan Standard. This offering provides automated scanning of a wide range of application security vulnerabilities with prioritized results. In addition to covering the top 10 vulnerabilities per Open Web Application Security Project, AppScan Standard automatically updates rules to keep current for the latest threats.

Going further

This short article covered a lot of ground, exploring static and dynamic testing as well as vulnerability scanning approaches and tools. The question is then, which is the right choice for your development? Some form of each should be applied across the development life cycle, during design and development, test, and in production. Whether you choose a collection of open source offerings or an integrated family of products such as IBM's AppScan portfolio, the choice is really between delivery of a reliable product and the next public, high-profile security failure.
Related topics

- Zero-day exploits are the holy grail of hacktivists and cyber-criminals and as such are in high demand. An underground economy exists for the purchase of such exploits, as illustrated in Shopping for Zero-Days: A Price List for Hackers' Secret Software Exploits. This article provides an interesting look at the brokering of exploits, illustrating the requirement for building secure software.
- Static and dynamic testing is a mature field with a growing list of tools and technologies. Wikipedia provides a useful description of each approach (including formal methods).
- The Splint tool, featured in this article, is a useful Lint utility for identifying potential issues through source scanning. Other useful static testing tools include:
  - codesake_dawn
  - VisualCodeGrepper
  - LAPSE+
  - FindBugs
  - RIPS
  - PyChecker
  - pylint
  - JSLint
  - JSHint
  - Brakeman
- IBM Security AppScan Source is a professional static testing tool that can be used to find vulnerabilities early in the SDLC. In addition to provide source scanning, AppScan Source fits within a security portfolio that includes integration with other security tools and provides useful reports for software compliance.
- IBM Security AppScan Standard is a professional application vulnerability scanner that offers not only the most advanced automated scanning capabilities but also highly accurate results. For large environments, AppScan Standard can recommend prioritization of remediation as a function of the vulnerabilities found, with clean integration with other products in the IBM security portfolio.
- The Valgrind dynamic testing tool is one approach to dynamic testing but not the only one. Other approaches include:
  - Pin
  - DynamoRIO
  - gcov
  - dmalloc
- I also explored ideas and tools around vulnerability testing, in particular the nmap utility. Vulnerability scanning and related tools are an increasingly popular area of testing for validating an application in execution. Other useful tools in this category include:
  - scanrand
  - metasploit
  - OpenVAS
  - Nikto
  - sqlmap
  - w3af
• Skipfish
• Wfuzz
• tcpdump
• Wireshark

© Copyright IBM Corporation 2013
Trademarks
(www.ibm.com/developerworks.ibm/trademarks/)