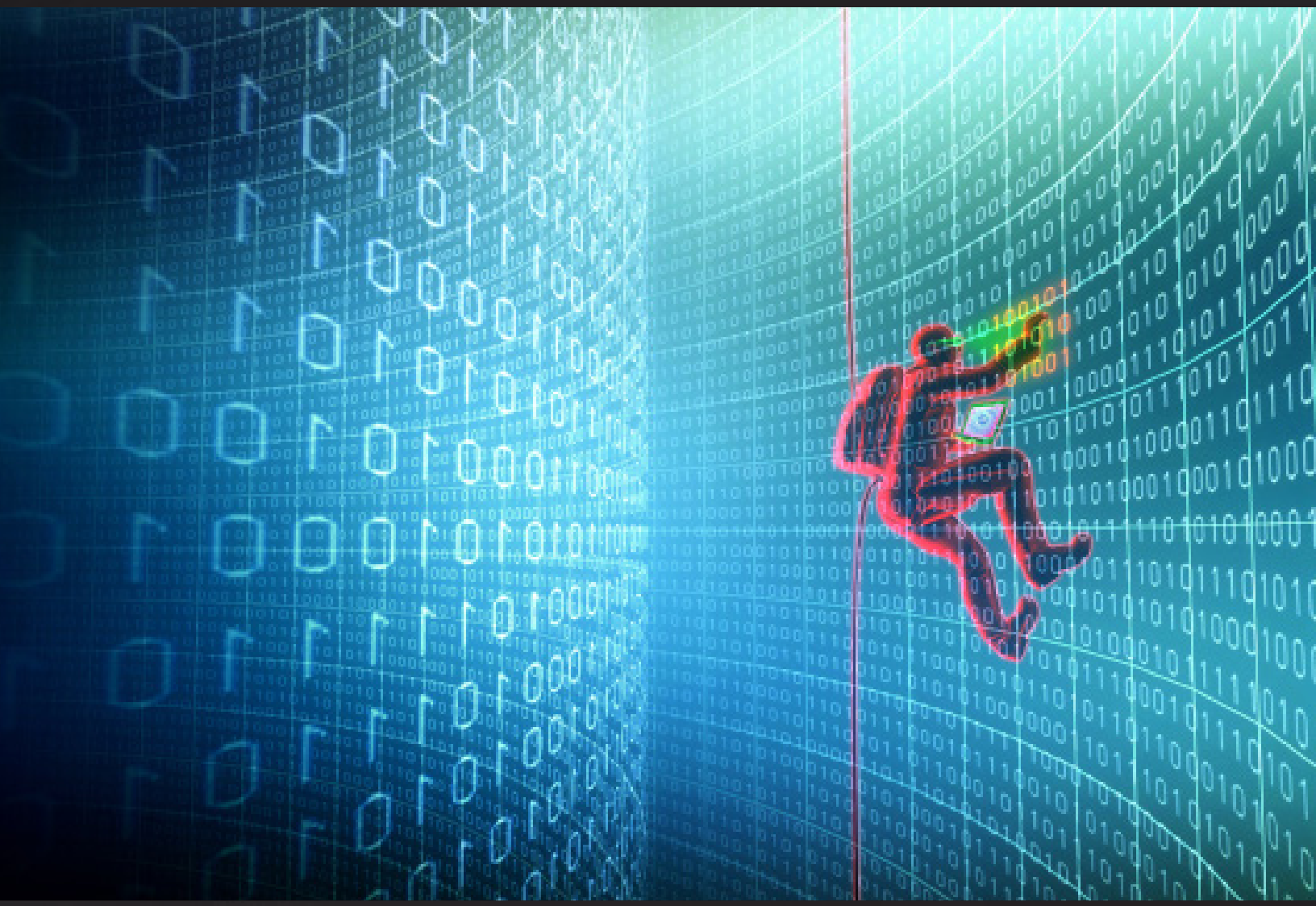
GRAMMATECH

PREVENT CYBERCRIME AND INSIDER ATTACKS

IN YOUR COMPANY WITH STATIC ANALYSIS



TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

The security threat posed by insiders is often underestimated. According to an IBM study, 32% of attackers are insiders and 24% are "inadvertent actors" (e.g. people making mistakes that lead to a system breach or incorrect behavior.) One such class of insider attack is malicious code added during development that allows for future exploitation. Advanced static analysis tools can detect these within source and binary code before they get shipped to customers. In addition to existing detection for security vulnerabilities, this paper also talks about specific security vulnerability checks to detect certain insider attacks.

WHAT ARE INSIDER ATTACKS?

Insiders are people working inside the secure perimeter either as users, developers or other trusted personnel. The big difference from regular cyber-attacks is the insider is often on a trusted network or has physical access to the device or system. The attack surface for insiders is larger than for outsiders. According to the SEI, 21% of electronic crime was perpetrated by insiders and 43% of respondents to their survey had experienced at least one insider attack.

Insider attacks might be due to unintentional mistakes or intentional malice by disgruntled employees. Attacks can be perpetrated when a product is in the field by intentional misuse or via pre-programmed vulnerabilities. Attacks programmed into the product ahead of time are of interest in this post, and in the same SEI survey, 37% of insider attacks were caused by "virus, worms or other malicious code."

THE ROLE OF STATIC ANALYSIS TOOLS IN IMPROVING SECURITY

Static analysis tools provide critical support in the coding and integration phases of development. Ensuring continuous code quality, both in the development and maintenance phases, greatly reduces the costs and risks of security and reliability issues in software. In particular, it provides some of the following benefits:

- Continuous source code quality and security assurance: Static analysis is often applied initially to a large codebase as part of its initial integration as discussed below. However, where it really shines is after an initial code quality and security baseline is established. As each new code block is written (file or function), it can be scanned by the static analysis tools, and developers can deal with the errors and warnings quickly and efficiently before checking code into the build system. Detecting errors and vulnerabilities (and maintaining secure coding standards, discussed below) in the source at the source (developers themselves) yields the biggest impact from the tools.

- Tainted data detection and analysis: Analysis of the data flows from sources (i.e. interfaces) to syncs (where data gets used in a program) is critical in detecting potential vulnerabilities from tainted data. Any input, whether from a user interface or network connection, if used

unchecked, is a potential security vulnerability. Many attacks are mounted by feeding specially-crafted data into inputs, designed to subvert the behavior of the target system. Unless data is verified to be acceptable both in length and content, it can be used to trigger error conditions or worse. Code injection and data leakage are possible outcomes of these attacks, which can have serious consequences.

- Third-party code assessment: Most projects are not greenfield development and require the use of existing code within a company or from a third party. Performing testing and dynamic analysis on a large existing codebase is hugely time consuming and may exceed the limits on the budget and schedule. Static analysis is particularly suited to analyzing large code bases and providing meaningful errors and warnings that indicate both security and quality issues. Binary code analysis can analyze binary-only libraries and provide similar reports as source analysis when source is not available. Ideally, binary analysis should work in a mixed source and binary mode to detect errors in the usage of external binary libraries from the source code.

- Secure coding standard enforcement: Static analysis tools analyze source syntax and can be used to enforce coding standards. Various code security guidelines are available such as SEI CERT C and Microsoft's Secure Coding Guidelines. Coding standards are good practice because they prevent risky code from becoming future vulnerabilities. As mentioned above, integrating these checks into the build and configuration management system improves the quality and security of code in the product.

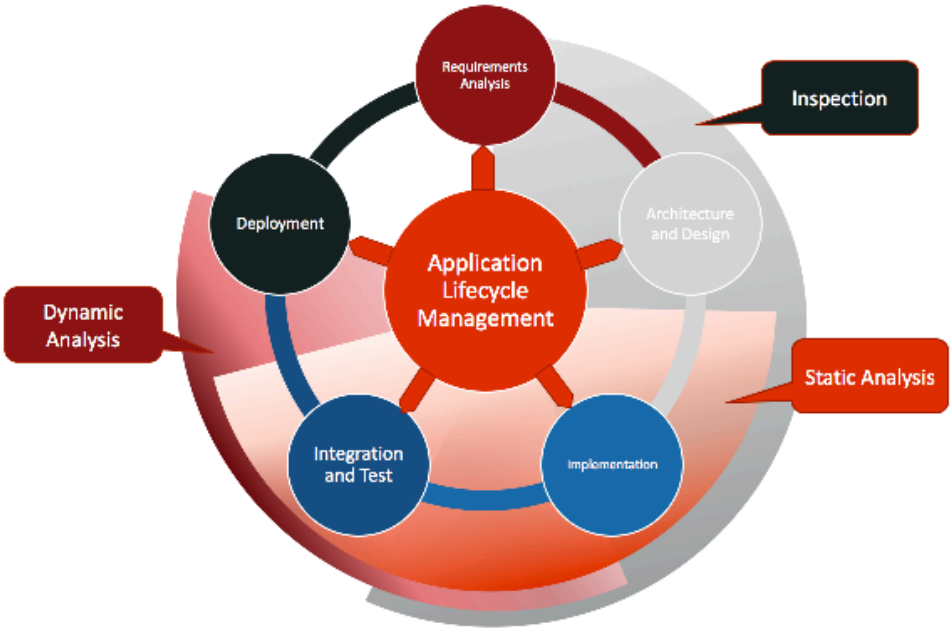


Figure 1: The role of static and dynamic analysis in a software development lifecycle.

BINARY ANALYSIS FOR FINDING ATTACKS

CodeSonar's binary code analysis technology is capable of analyzing stripped optimized executables; roughly speaking, it finds the same class of defects that can be found in the source code. The tool's new integrated analysis is capable of analyzing source and binaries simultaneously. This is useful in cases where you have source code for most of the project, but only binary libraries for some components.

Although the possibility of investigating and fixing issues found in third-party code is often limited, binary analysis does provide a bellwether of the quality and security of the code. Customers of commercial off-the-shelf (COTS) products can go back to technical support of the vendor and ask for confirmation and analysis of the discovered vulnerabilities. The key here is that the product vulnerability is better understood -- third-party software with a large number of security issues found using binary analysis must be dealt with appropriately either internally or through negotiation with a software vendor.

BINARY AND SOURCE HYBRID ANALYSIS

Binary analysis really shines when used in a hybrid fashion with source analysis. Source static analysis has much more information about the intent and design of the software than binary analysis. However, whenever an external library is called, including standard C/C++ libraries, static analysis can't tell if the use of the function is correct or not (assumptions are made, of course, for well-known functions like strcpy()). By combining source and binary analysis, a more complete analysis is possible. For example, if an external function takes a pointer to a buffer and a buffer overflow is possible with misused parameters, hybrid static analysis can detect this problem.

GrammaTech CodeSonar provides additional malicious code detection for detecting dangerous process creation, use of CHROOT, and possible time bombs. The following examples illustrate the types of exploits CodeSonar checks for.

EXAMPLES OF POTENTIAL INSIDER ATTACK TYPES DETECTED BY GRAMMATECH CODESONAR

GrammaTech CodeSonar provides additional malicious code detection for detecting various dangerous code constructs. The following examples illustrate the types of exploits CodeSonar checks for.

UNTRUSTED PROCESS CREATION

Process creation is always potentially dangerous, especially if it's possible to manipulate the process name or parameters (e.g. command injection.) CodeSonar detects untrusted process creation by checking the arguments to functions that create processes against a blacklist. The blacklist is configurable but includes well known, potentially dangerous commands. The example below

produces a warning with the sh command but not with “myprocess” which isn’t in the blacklist.

```
#include <stdlib.h>
#include <stdio.h>

void ut_proc(const char *command) {

    FILE *pipe_file;

    if (pipe_file = (FILE*)popen(“/usr/bin/myprocess”,”r”)) {
      /* not blacklisted */
      pclose(pipe_file);
    }

    if (pipe_file = (FILE*)popen(“/usr/bin/sh”,”r”)) {
      /*’Untrusted Process Creation’ warning issued here */
      pclose(pipe_file);
    }

}
```

UNTRUSTED LIBRARY LOAD

An insider might try to load an untrusted dynamic library at runtime that contains malicious code. However, detecting every library load would cause too many false positive warnings so a blacklist of unwanted libraries is recommended (regular expressions are supported.)

In CodeSonar the project configuration file would have the following, for example:

```
UNTRUSTED_LIB_BLACKLIST += ^.*hack.*$
```

CodeSonar will issue a “Untrusted Library Load” warning in the following code:

```
#include <lfcn.h>

void * io_ut_lib_bad(void) {
  return dlopen(“./myhackylibrary.so”, RTLD_LAZY);
  /* ‘Untrusted Library Load’ warning issued here */
}
```

```
void * io_ut_lib_ok(void) {
  return dlopen(“”./myproperlibrary.so”, RTLD_LAZY);
  /* ok: does not include blacklisted substring “hack */
}
```

CHROOT WITHOUT CHDIR

Issuing the `chroot()` (change process root directory) Unix/Linux command is potentially dangerous, and malicious code can exploit the situation to access files in other parts of the system. A best practice is to issue the `chdir()` (change current process directory) command right before or after `chroot()`. In the example below, a warning is raised because `chdir()` isn't always called right after the `chroot()` command due to the check on the variable `fname`.

```
#include <unistd.h>
#include <stdio.h>
int chroot_no_chdir(const char *fname, char *buf) {

    FILE *localfile;
    int bytesread=0;

    if (chroot(“/downloaddir”) == -1) {
      /* chroot without chdir’ warning issued here */
      return 0;
    }
    if (fname) {
        if (localfile = fopen(fname, “r”)) {
            bytesread = fread(buf, 1, sizeof(buf), localfile);
            fclose(localfile);
        }

        if (chdir(“/”)==-1) {
          /* chdir() is only called if fname!=NULL */
          return 0-bytesread;
        }
     }
     return bytesread;

}
```

POTENTIAL TIME BOMB

Time bombs are possible in code that uses and checks for time values from the system clock. Reasonable checks for time are acceptable, but code that uses a value not derived from the current time, such as a hardcoded constant, may be "waiting" for a specific time and date to execute malicious code. In the example below, the second "if" statement compares the time value against a hardcoded constant.

```
#include <time.h>

void misc_timebomb(void) {

    time_t deadline = 1893456000;
    time_t now = time(NULL);

    if (now > time(NULL)) {
      /* ok: time value compared against another time value */
      /* ... */
    }
    if (now < deadline) {
      /* 'Potential Timebomb' warning issued: time value compared */
      /* against non-time value */
      return;
    }
    /* An inside attacker could put malicious code here:
     * it would only be executed once the deadline was past. */

}
```

UNTRUSTED NETWORK ADDRESSES OR PORTS

An insider could allow external access to an application or device via a network connection or through an unauthorized network port. Intended network connections may be indistinguishable from illegitimate ones under casual inspection, more so if addresses and port numbers are purposely obfuscated. Static analysis tools can easily detect network connection functions. However, to prevent false positives, specifying a list of addresses and port numbers to exclude is recommended. For example, in CodeSonar the project configuration file would contain the following:

```
NETWORK_HOST_BLACKLIST += allow ^0:0:0:0:0:0:0:1$

NETWORK_HOST_BLACKLIST += .+\.[a-zA-Z]{2,6}($|\s+|\\|/|:)
```

For network ports:

```
NETWORK_PORT_WHITELIST += ^80$
```

CodeSonar will issue an “Unstrusted Network Host” warning in the following code.

```
#include <stdlib.h>
int ut_host() {
  int status;
  struct addrinfo *res;

  status = getaddrinfo(“0:0:0:0:0:0:0:1”, “80”, NULL, &res);
  /* explicit blacklist exception */
  if (status != 0) {
    if (res)
      free(res);
    return status;
  }

  status = getaddrinfo(“2001:DB8:1:2:3:4:5:6”, “80”, NULL, &res);
  /* ‘Untrusted Network Host’ warning issued here. */

  if (status != 0) {
    if (res)
      free(res);
    return status;
  }
  /* ... */
}
```

CodeSonar will issue a “Untrusted Network Port” warning in the following code:

```
#include <stdlib.h>

void ut_port( const char *myhost, char *myport ) {
  struct addrinfo *res;
  int rv;

  rv = getaddrinfo( myhost, “1234”, NULL, &res );
  /* ‘Untrusted Network Port’ warning issued here */
```

```
  rv = getaddrinfo( myhost, “80”, NULL, &res );
  /* whitelisted */
  /* ... */
}
```

WEAK CRYPTOGRAPHIC FUNCTIONS

The use of weak cryptographic functions is poor security. However, an insider may intentionally use these functions to overcome the encryption at a later date. For example, application files with MD5 or DES encryption might seem adequate, however these are known weak encryption algorithms. Sensitive data could be decrypted by attackers with access to the stored files.

CodeSonar issues a “Weak Cryptography” warning with the following code:

```
#include <openssl/md2.h>

void weakcrypto(MD2_CTX *ctx) {
  if (MD2_Init(ctx)) {
     /* ‘Weak Cryptography’ warning issued here */
     /* ... */
  }
  /* ... */
}
```

SUMMARY

Insider threats in the form of malicious code written by insiders are a significant, but often overlooked, source of cyberattacks. Advanced static analysis tools can detect intentional malicious code using security vulnerability analysis. GrammaTech CodeSonar is an advanced static analysis tool that detects different types of malicious code, including potential inside attacks, and other security vulnerabilities.

REFERENCES

IBM X-Force® Research 2016 Cyber Security Intelligence Index
2011 CyberSecurity Watch Survey: How Bad Is the Threat?
Common Sense Guide to Mitigating Insider Threats 4th Edition
Eliminating Vulnerabilities in Third-Party Code with Binary Analysis
Protecting Against Tainted Data in Embedded Apps with Static Analysis

GrammaTech, Inc. is a leading developer of software-assurance tools and advanced cyber-security solutions. GrammaTech helps organizations develop and release high quality software, free of harmful defects that cause system failures, enable data breaches, and increase corporate liabilities in today’s connected world. GrammaTech’s CodeSonar is used by embedded developers worldwide.