Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes

Barton P. Miller, Mihai Christodorescu, Robert Iverson
Tevfik Kosar, Alexander Mirgorodskii, Florentina Popovici

Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{bart,mihai,riverson,kosart,mirg,pif}@cs.wisc.edu

Abstract

Programs running on insecure or malicious hosts have often been cited as ripe targets for security attacks. The enabling technology for these attacks is the ability to easily analyze and control the running program. Dynamic instrumentation provides the necessary technology for this analysis and control. As embodied in the DynInst API library, dynamic instrumentation allows easy construction of tools that can: (1) inspect a running process, obtaining structural information about the program; (2) control the execution of the program, (3) cause new libraries to be dynamically loaded into the process' address space; (4) splice new code sequences into the running program and remove them; and (5) replace individual call instructions or entire functions.

With this technology, we have provided two demonstrations of its use: exposing vulnerabilities in a distributed scheduling system (Condor), and bypassing access to a license server by a word processor (Framemaker). The first demonstration shows the danger of remote execution of a job on a system of unknown pedigree, and the second demonstration shows the vulnerabilities of software license protection schemes. While these types of vulnerabilities have long been speculated, we show how, with the right tool (the DynInst API), they can be easily accomplished. Along with this discussion of vulnerabilities, we also discuss strategies for compensating for them.

1 Introduction

Programs in execution have long been considered to be immutable objects. Object code and libraries are emitted by the compiler, linked and then executed; any changes to the program require revisiting the compile or link steps. In contrast, we consider a running program to be an object that can be examined, instrumented, and re-arranged on the fly. The DynInst API provides a portable library for tool builders to construct tools that operate on a running program. Where previous tools might have required a special compiler, linker, or run-time library, tools based on DynInst can operate directly on unmodified binary programs during execution. In this papers, we show how this technology can be used to subvert system security. The discussions will be based on two example cases: exposing vulnerabilities in a distributed scheduling system (Condor), and bypassing access to a license server by a word processor (Framemaker).

For the Condor study, we create "lurker" processes that can be left latent on a host in the Condor pool. These lurker processes lie in wait for subsequent Condor jobs to arrive on the infected host. The lurker will then use Dyninst to attach to the newly-arrived victim job and take control. Once in control, the lurker can cause the victim job to make requests back to its home host, causing it execute a wide variety of system calls.

For the license-server study, we constructed a collection of Dyninst-based tools that allowed us to understand the control flow within the application (Framemaker) program. As a result, we were able to detect and remove Framemaker's contact with the license server. In addition, there are frequent checks within Framemaker to see if it has cached a valid license credential. Using our Dyninst-based tools, we were able to locate and neutralize these checks.

For each of these studies, we provide some suggestions as to how to make them less vulnerable to attack.

2 DynInst API

DynInst is an architecture-independent library for making on-the-fly modifications to a running program. It does not require any special preparation of the executable such as re-compiling or re-linking. Figure 1 shows the organization of a software tool based on the DynInst API. The tool, called the *mutator*, is linked with the DynInst API library and makes API calls to control and modify the application program, called the *mutatee*. The DynInst library attaches to the mutatee with the usual process debugging interface provided by the operating system, such as ptrace or `/proc` on Unix or the process control API on Windows/NT. Some operations, such as reading and writing the memory of the mutatee, are performed by DynInst directly through this interface. Other more complex operations, such as allocating memory, are executed by a library installed by DynInst in the mutatee, called the *run-time instrumentation library* (RTinst).

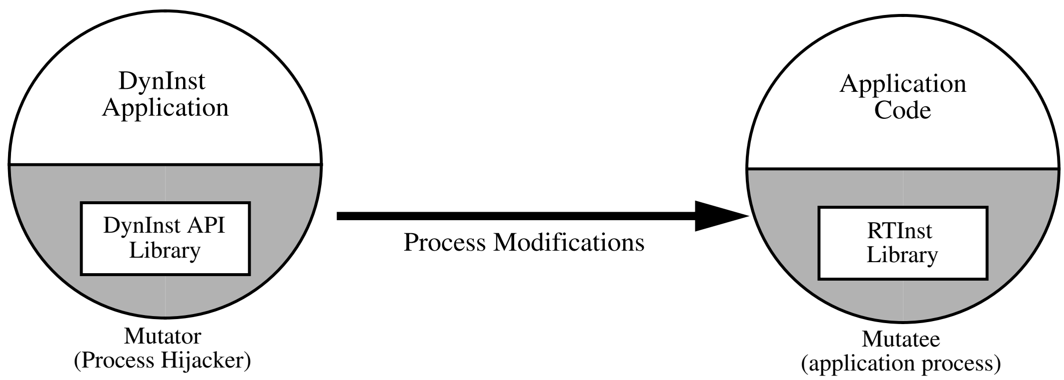


Figure 1: DynInst API Operation

With DynInst, the mutator can splice *code patches*, sequences of machine instructions, at most locations in the mutatee. The mutator is provided with structural information, including the application call graph, intraprocedural control glow graphs, and a list of function entry points, exit points, and call sites. DynInst provides an architecture-independent mechanism for specifying code patches in terms of familiar program data and control flow operations, including assignment, logic and arithmetic, branching, and function calls. Individual function calls in the mutatee can be replaced with calls to other functions (existing or newly loaded into the program; all calls to a function also can be replaced. The mutator can cause the mutatee dynamically load new libraries (.dll or .so) at any time. It also can make an inferior RPC into the mutatee to cause it to asynchronously execute any function in the mutatee, including functions that are part of the mutatee's code and functions in libraries that were loaded by DynInst.

3 Attack 1: The Lurking Condor Job

Our first attack demonstrates the use of DynInst technology to expose security vulnerabilities in a distributed computing environment.

3.1 Background

Condor is a system that allows users to schedule and run application programs on idle hosts in a widely-distributed environment [7,8,9]. Condor users do not have to have user accounts and privileges on theses hosts; Condor takes responsibility for running the application programs in such a way that they should do no harm to the machines on which they run. However, this remote execution idiom can expose the Condor user to security risks.

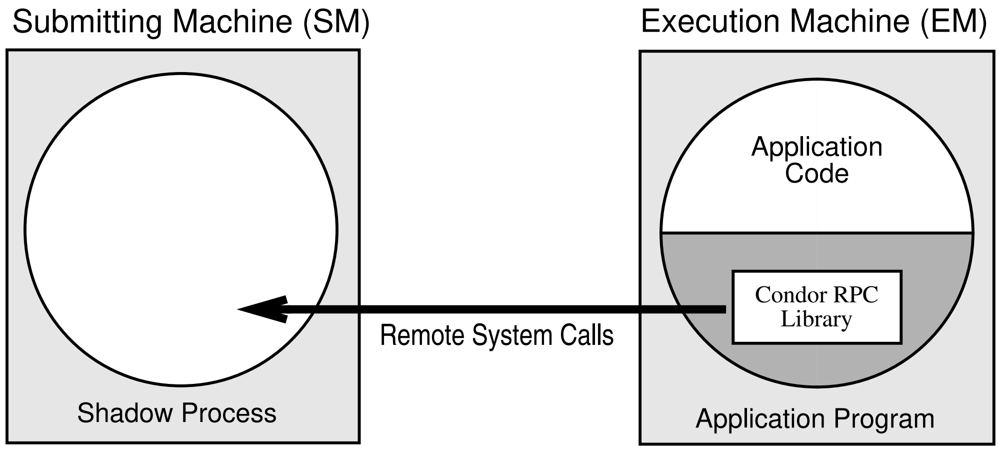


Figure 2: A Condor Job in Execution
The application program is linked with the Condor RPC library, causing its system calls to be sent from the Execution Machine (EM) to be processed on the Submitting Machine (SM).

A Condor application is typically linked with a special version of the system-call library; this library replaces the standard system calls with RPC stubs that forward the calls back to the user's *submitting machine* (SM). When Condor starts a program on the remote *Execution Machine* (EM), it also starts a *shadow process* on the SM that receives remote system calls from the application program and executes with the normal privileges of the submitting user (see Figure 2). This remote system call path back to the SM provides a new type of security threat: someone on the remote machine might subvert the application program and cause it to make inappropriate and malicious requests to the user's SM. These requests might include accessing, modifying, or deleting private files, originating e-mail, proliferating a virus, or initiating password cracking software. Since this path to the SM could provide an attacker with access inside of an organization's firewall, "crunchy" environments (crispy on the outside, soft on the inside) are vulnerable.

Suppose a user has submitted a lawful job and it is scheduled on a malicious EM. Then the malicious superuser at the EM can gain control over the Condor job. In particular, the superuser is able to modify the image of the job at run time and make it execute arbitrary system calls. Even worse, as we demonstrate, is the ability of a normal Condor user to submit a malicious job that, at some later time, takes control over some other user's application program and cause it to make inappropriate system call requests. The remote execution scenario is not unique to Condor; it also occurs when a Web browser down-loads a Java applet. The applet often contacts the server (in fact, it is typically constrained to communicate only with its originating server) to make queries and requests. If we can easily subvert the applet, we might cause the server to perform inappropriate actions [*cite?*].

The key concept is that the DynInst library makes it easy to take control of another program, analyze its execution, and cause it to execute arbitrary instructions. Clever applications of this technology can avoid the need for special privileges and can make it difficult to trace the source of the intrusion.

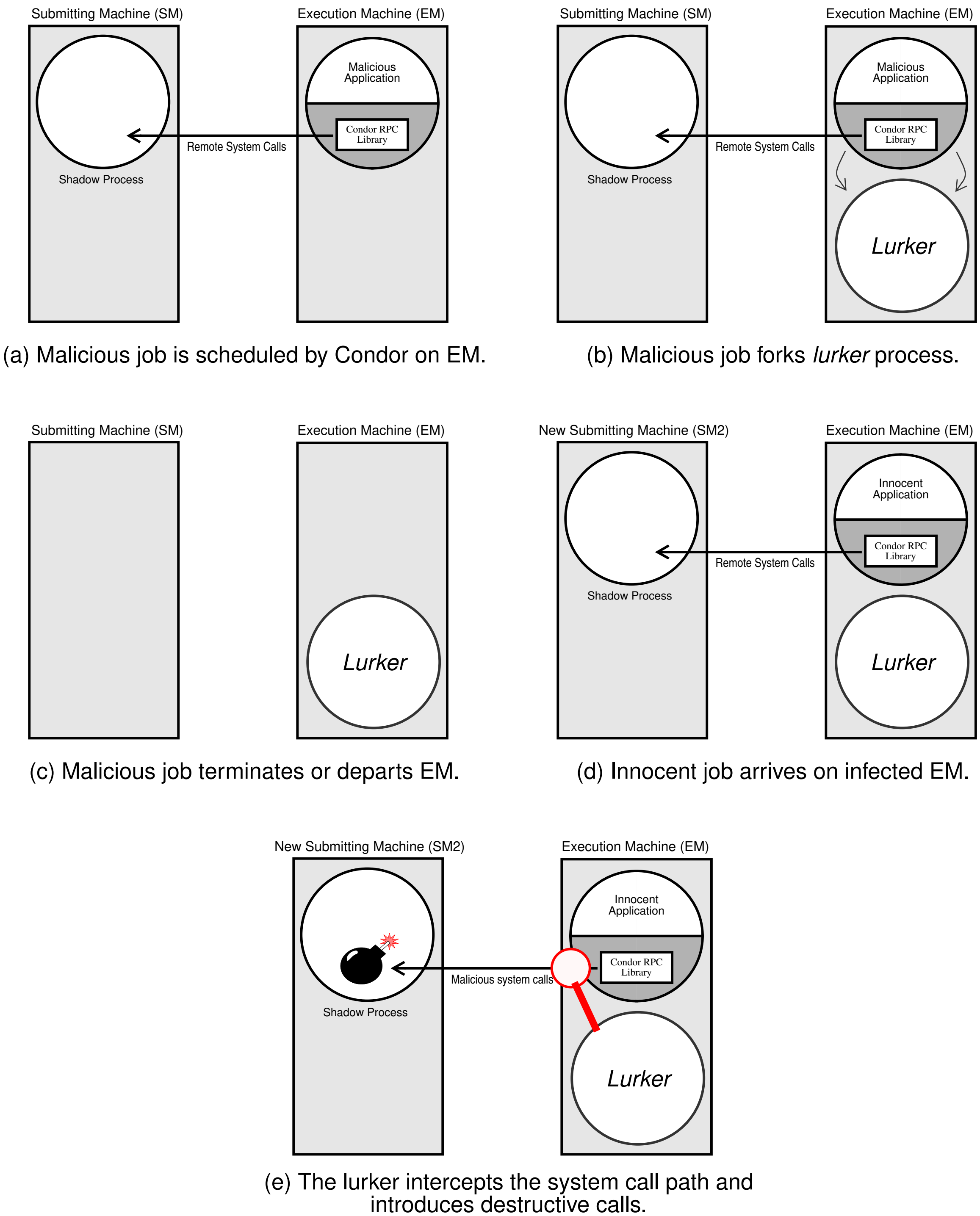
3.2 The Attack Strategy

The steps taken to subvert a Condor job are as follows:

1. Submit our malicious job to Condor through the normal submissions process.
2. At some time in the future, Condor will schedule the malicious job on an idle computer (Figure 3a). Note that this computer might be in the same organization as the submitting user, or one in a different Condor *flock*, located in another organization (possibly geographically far away). In a common Condor configuration, the job will run

with an anonymous, restricted user ID (typically “nobody”).

3. The malicious job creates a new process (forks). We call this process the *lurker* (Figure 3b).
4. The malicious job departs, either completing or being checkpointed and migrated by Condor (Figure 3c).
5. The lurker process will wait until an innocent Condor job arrives (Figure 3d).
6. Since the newly arrived job has the same user ID, the lurker attaches to the new job, intercepts the system calls, and causes inappropriate calls to be sent to the home node of the innocent job (Figure 3e).



(a) Malicious job is scheduled by Condor on EM.

(b) Malicious job forks *lurker* process.

(c) Malicious job terminates or departs EM.

(d) Innocent job arrives on infected EM.

(e) The lurker intercepts the system call path and introduces destructive calls.

Figure 3: Creating a Lurking Process to Take over a Innocent Condor Job

Some notes:

- The malicious job could be designed to run for a long period of time, likely resulting in checkpointing and migration multiple times. At each subsequent host it visits, it has the option of creating another lurker process. When the malicious job terminates or migrates, the lurker process is left running, presumably dormant.
- The lurker process could also lie idle for a long time before springing into action (after many subsequent Condor jobs had arrived and departed), making difficult to attribute the authorship of the malicious job.

We demonstrated the use of DynInst to create malicious lurking jobs. Our job was compiled to run on x86 Linux hosts and tested on an isolated Condor pool on one of our local clusters. The malicious job simply waited for an innocent job, attached to it, and caused it to modify files in the innocent user's home directory (we added entries to their `.rhosts` and `.k5login` files to allow an intruder free access to login into their host).

3.3 Protecting the SM from the Lurker Attacks

It is clear that there several approaches that the Condor can use to add the security of the SM. The most basic approach is to create a "sandbox" around the shadow process on the SM. Sandboxing techniques can include:

- Restrict the system calls that the shadow will accept. The restricted calls might include process creation (fork and exec), network access (socket), and process control (kill).
- Restrict access to particular file directories. This restriction could be simple and severe, by using chroot to limit the shadow to access only the current directory. More flexible policies and possibly per-job policies also could be used.

Security on the EM machine can also be increased. Techniques useful on the EM include:

- Some UNIX versions, such as Sun's Solaris, allow you to specify the set of allowable system calls to be specified when a process is created. When Condor starts a job on the EM, it would disallow such calls as fork, preventing the creation of lurking processes. It is necessary for the operating system kernel to perform this restriction. While the Condor remote system call library does not allow a job to call such functions as fork, the job can simple create its own assembly language version of the call.
- Clean up after a job migrates or completes. Condor can check for all new processes created under the "nobody" user ID and remove these when it thinks that the job is done on this host. Any lurker processes should be caught by this strategy.
- User multiple user ID's. If Condor had a relatively large collection of user ID's, and cycled among these ID's when starting new jobs, then a lurking process would not have access to subsequently arriving jobs (until enough jobs were run that Condor cycled through the list). This technique is less desirable to Condor because it requires an organization to add new user IDs on each machine in the Condor pool (current UNIX platforms have, as a default, a standard "nobody" user so no privileged per-host administration is necessary).

The above three techniques would be effective if the EM host is trusted. If EM has a malicious owner (with privileged access), then they can bypass any of these techniques. In general, running a job on a malicious host is problematic; this is a topic that we are currently researching).

4 Attack 2: Subverting License Checking

Our second attack demonstrates the use of DynInst technology to subvert commercial software license protection.

4.1 Background

Many modern software products use some form of checking that the user is legally authorized to run the program.

Such checks are intended to prevent piracy and enforce the software vendor's product licensing terms. A common approach to license checking involves getting some data from an external source (such as a protected file or a secured network server) and verifying it for validity.

Our goal was to bypass such checking and attain full program functionality even when the license data could not be obtained. We developed several DynInst-based tools that help analyze a running program without any prior information about the executable and without access to source code. Our target application was the Framemaker word processing tool from Adobe, and these tests were performed on an UltraSPARC IIi running Solaris.

While w e focused on programs that use a license server, our results can be easily extended to products using local license files. The evaluation of the application is similar in both cases since the access to the remote or stored credential often appears in a similar, well-defined place in the program.

4.2 The Attack Strategy

We approached the problem of bypassing license checking from several angles. First, we attempted to see the program as a black box, from which we capture the I/O for later replay. Second, we traced the flow control within the program to understand where the license checking is performed. The combination of these two methods helped us understand the program behavior, a necessary step in modifying the license checking.

We capture I/O operations by using DynInst to attach itself to the mutatee and trace all library functions that performs I/O, thus making it applicable in analyzing the behavior of programs with network communications. We replace the open, read, write, send, and recv library function, among others, with custom versions. This new open sets up a trace file for later use by read and write, which are modified to copy their data into the mirror file. In this way we can save the contents of any temporary files, socket activity, and data worth analyzing. Using this technique, we located those I/O operations that were specific to contacting the license server.

Once we were able to trace the I/O operations of the application, we then traced the control flow (at the function call level) within the application. By tracing the control flow for both the cases where the application was able to successfully contact the license server and the case where it failed, we were able to locate where the license checking occurs. More important, we were able to determine the functions to be skipped or replaced to avoid the failure of the license check.

4.3 The Attack Toolkit

Our attack techniques are embodied in a collection of tools that we built. These tools simplified the analysis task and are generally useful for other similar tasks. The tools include a function call tracer, function argument parser, and Java to DynInst compiler. These tools can be accessed through two different interfaces, the DynInst command line interpreter or the License Bypasser GUI.

Function Call Tracer

Using DynInst, tracing is easy: we insert code at the beginning and at the end of a function in the mutatee. The inserted code will generate output that the mutator can interpret. The depth of the call stack, order of calls, and return values from each function are reported. To avoid inserting excessive instrumentation, we incrementally instrumented the program, following the calls down the call stack, as follows. Starting with main, we:

1. Before calling the function: insert a trap at the entry and exit points of the function.
2. On entry to an instrumented function: insert a trap before and after all the function calls made by that function.
3. On exit from a function: remove the traps before and after the calls made by that function.
4. After the call to the function completes: remove the entry and exit traps.

In step 2, if we discover a call that uses a function pointer, we cannot immediately identify the destination of the call. In this case, we instrument the call site to determine the address of the callee. Once we have that address, we use the DynInst library to map that address to the function name. The result is that we eventually generate a complete call graph for the application. This approach is similar to the one used by the DynInst-based Paradyn performance tools, when it automatically instruments an application program in searching for performance problems [2].

Function Argument Parser

The function argument parser makes it easy to track the type and name of each parameter to a function in the application, if either the program was compiled with debugging information or the user provides this information. We developed a tool that helps automate this process by parsing a strings such as:

```
int open(char *name, int oflags, int mode)
char *strcpy(char*, char*)
```

to determine each of the arguments. This information is passed to a code-snippet generator that will produce DynInst API calls to generate instrumentation code. The instrumentation code, when inserted into the application, will collect the parameter and return values to provide a more detailed analysis of the internal workings of the application program.

Java to DynInst Compiler (JavaD)

To simplify the task of creating code snippets, we built a Java to DynInst compiler. The DynInst API calls operate at the machine-language level for building code sequences. While this is a simple interface, it can be cumbersome and error-prone to manually construct these sequences.

The Java to DynInst compiler (JavaD) supports all the arithmetic expressions, function calls and if statements that are necessary for snippet insertion. Since there are no loops in DynInst, these constructs were not included in our compiler. Figure 4 shows an example of a Java snippet that conditionally opens a file, and its corresponding sequence of DynInst calls.

```
class X {
  public int open(String path, int flag, int mode) { }
  public static void main(String argv[]) {
    int test = 1;
    if (test != 0) {
      open(filename, O_WRONLY | O_CREAT, 0666);
    }
  }
}
```

Figure 4: Java Input to the JavaD Compiler and the Corresponding DynInst API Output

```
BPatch_function *openFunc = $IMAGE->findFunction("open");
BPatch_Vector<BPatch_snippet*> openArgs;

BPatch_variableExpr *test = $THREAD->malloc(*$IMAGE->findType("int"));

BPatch_arithExpr $ArithExpr(BPatch_assign, *test,
                            BPatch_constExpr(1)); //<-Arith. Expr.

BPatch_constExpr path(filename);
BPatch_constExpr flag(O_WRONLY|O_CREAT);
BPatch_constExpr mode(0666);

openArgs.push_back(&path);
openArgs.push_back(&flag);
openArgs.push_back(&mode);

BPatch_funcCallExpr openCall(*openFunc, openArgs);      //<-function call

BPatch_Vector<BPatch_snippet*> StatementsInsideIF;
StatementsInsideIF.push_back(&/*pointer to the statement goes here*/);
BPatch_sequence IFSequence(StatementsInsideIF);
BPatch_boolExpr boolExpr(BPatch_eq, *test, BPatch_constExpr(0));
BPatch_ifExpr IFExpr(boolExpr,IFSequence);      //<-If Epression
```

Figure 4: Java Input to the JavaD Compiler and the Corresponding DynInst API Output

DynInst Command Line Interpreter (Dynit)

The DynInst command line interpreter, called dynit, provides access to the previously described analysis tools and the DynInst code generation primitives; it also includes various debugger-like process control commands. Dynit provides a scripting facility that aids in performing repeated experiments.

The Dynit code generation functions provide a command line interface to the basic code snippet construction and insertion primitives. These snippet definitions can be constructed and re-used. Code insertion include simple insertion, and function call and whole function replacement. The process control commands allow the user to start, stop, insert breakpoints, control tracing, and inspect (print) function state information.

The License Bypasser GUI

The License ByPasser GUI is our top level tool for accessing the tool set. It allows the user to walk through the call graph of the target application, search it, list functions that call or are called from a specific function, list modules and license-related functions in the application, load user-defined libraries, replace function calls and continue execution. All the dynit commands are accessible through the ByPasser GUI. Figure 5 shows part of the GUI.

4.4 Attacking Framemaker

We first traced the network I/O from the Framemaker application, both when allowed to successfully contact a license server, and then prevented from making successful contact.

We next traced the control flow in Framemaker, using the previously described tools. We found that there are 86 functions that directly perform some kind of license-related activity and, by transitivity, these functions (and functions called from these function) call a total 1,181 functions, out of a total of 16,943 functions in FrameMaker.

By inspecting these top level functions, we discovered the following characteristics of Framemaker, relative to license checking:

- During the initialization phase, FrameMaker's main function calls NlOpenLicenses, which contacts the license

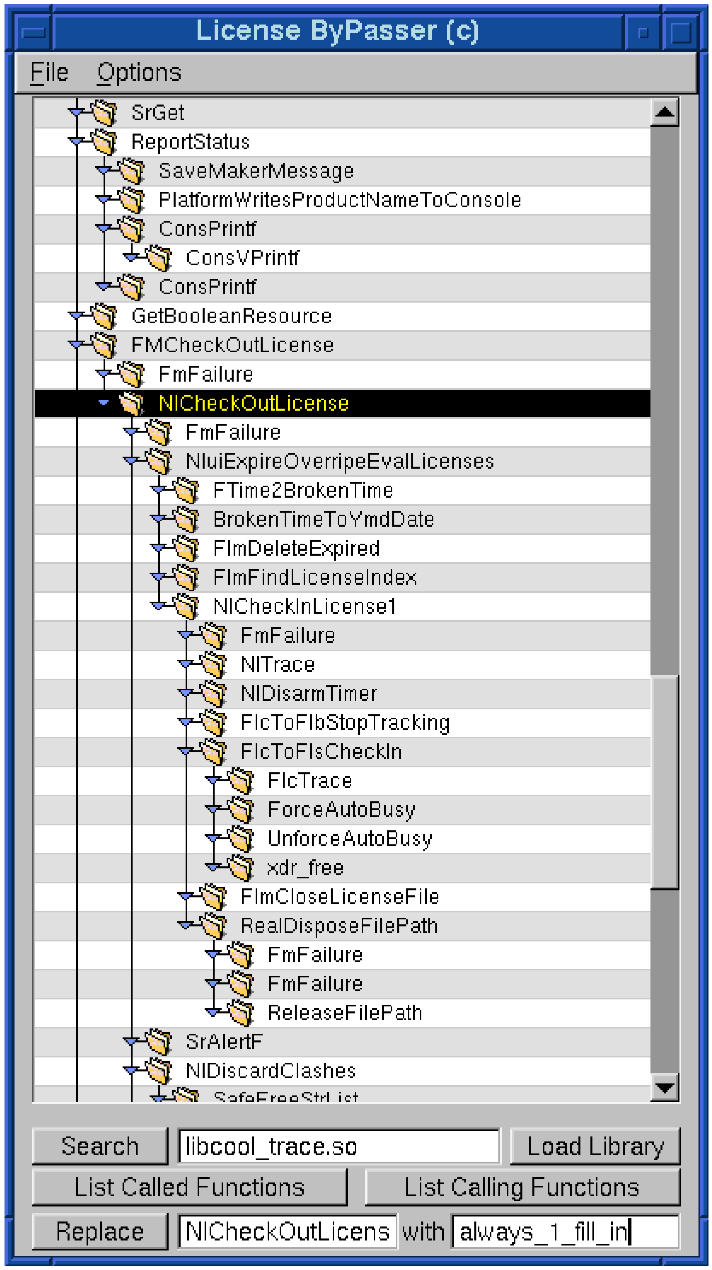


Figure 5: Interface to the License Bypasser GUI

server, retrieves the license data, and stores it in memory for later use. Even if the NlOpenLicenses fails and no license is retrieved, the program continues initializing, going through all the steps for setting up windows, reading the user defaults, the Most-Recently-Used document list, and other configuration settings.

- At the end of the initialization phase, main calls NluiCheckLicense. This function checks the license data in memory. If no valid license was obtained, FrameMaker displays a dialog box asking the user whether to go into “demo” mode or exit (demo mode does not allow the user to save files). It then calls ChangeProductToDemo or FmExit, depending on the user input.
- The program continues execution and displays the standard FrameMaker tool bar. The license checking validation is performed once more, when the user opens or create a document. If the license data is missing or invalid, text warning about the lack of a license is displayed, and access to the full functionality is denied. Otherwise, full

functionality is enabled, and the user can proceed with creating, editing, and saving documents. The license checking validation code is called frequently while the user edits a document, even though an initial license check validation was done.

We successfully bypassed the licensing checks by the following steps. Each of these steps was facilitated by using DynInst to modify the executing Framemaker application.

1. Allowed the retrieval of the license data to fail.
2. Prevented FrameMaker from entering demo mode by deleting the function call of ChangeProductToDemo.
3. Bypassed the first license data validation by skipping over the sequence of code that performed it.
4. Modified all later license data validations to always succeed, regardless of the presence of the license data in memory. The modification is done by changing NluiCheckLicense to always return "true".

Using this controlled failure mode, we successfully ran FrameMaker without a license being obtained from the license server.

4.5 Protecting the Application from License Attacks

In many ways, good software engineering practices make it easier to find the parts of an application related to checking licenses. The common functions of obtaining a license and verifying its validity were encapsulated in functions and used consistently throughout the application. Such clean design is usually intended to make software more reliable, easier to maintain, and easier to incorporate new functionality. These same characteristics help the person who intends on circumventing the checks.

Basic code obfuscation techniques [3] can be used to make this type of checking more difficult. These techniques can include obscure naming of modules and functions and violating modularity by having many implementations of the same functionality. The multiple implementations should certainly include the license check function, but should also include the error reporting code. If the checking code detects a violation, it will then report an error. It is a simple task to detect all calls to the error reporting code, and list the stack-trace showing which functions reported the error.

5 Conclusions

The goal of this paper was to provide concrete examples of the kinds of problems that have caused speculations for many years. It is easy to monitor and control almost any running program. It is also easy to make arbitrary changes that program's behavior. The DynInst library makes this type of activity a commodity. The Condor and Framemaker examples were intended to show specific ways in which such techniques could be applied to production software.

There is always a tension in the decision as whether it is better to reveal a security problem, thereby raising awareness and providing strong incentive to try to fix the problem, or to keep such problems a secret. Unfortunately, it is usually the innocent users of a software application or system that are the last to know about such vulnerabilities. We believe that openness of such problems raises the general awareness of the community and causes a general improvement in security.

The techniques used in this paper present serious challenges to the security community. Safe remote execution, including preventing inappropriate operations and prevent undetected modification or spoiling of computational results, is a difficult problem that needs more significant study. The problem of providing selective authorization to run an application program is similarly challenging. Both of these problems present scientific and commercial concerns, and both are likely to spawn interest research in the coming years.

References

[1] B. R. Buck and J.K. Hollingsworth, "An API for Runtime Code Patching", *Journal of High Performance Computing Applications* **14**, 4, Winter 2000, pp. 317-329.

[2] H.W. Cain, B.P. Miller, and B.J.N. Wylie, "A Callgraph-Based Search Strategy for Automated Performance Diagnosis", *Euro-Par 2000*, Munich, Germany, August 2000.

[3] C. Collberg and C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", *ACM Symposium on the Principles of Programming Languages (POPL)*, January 1998.

[4] J.K. Hollingsworth, B.P. Miller and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools", *Scalable High Performance Computing Conference,* Knoxville, May 1994.

[5] J.K Hollingsworth and B.P. Miller, "Using Cost to Control Instrumentation Overhead", *Theoretical Computer Science* **196***, 1-2* (April 1998). Invited paper.

[6] J.K Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *1997 International Conference on Parallel Architectures and Compilation Techniques*, November 1997, San Francisco, California.

[7] M. Litzkow, M. Livny, and M. Mutka. "Condor - A Hunter of Idle Workstations", *8th International Conference of Distributed Computing Systems*, San Jose, California, June, 1988.

[8] M. Litzkow and M. Livny. "Experience With The Condor Distributed Batch System", *IEEE Workshop on Experimental Distributed Systems*, Huntsville, Al., Oct. 1990

[9] M. Livny, J. Basney, R. Raman, and T. Tannenbaum., "Mechanisms for High Throughput Computing", *SPEEDUP Journal* **11**, 1, June 1997.

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools", *IEEE Computer* **28,** 11, (November 1995). Special issue on performance evaluation tools for parallel and distributed computer systems.

[11] V. Zandy, B.P. Miller, and M. Livny, "Process Hijacking", *8th IEEE International Symposium on High Performance Distributed Computing*, Los Angeles, August 1999.