Automatic Detection of Unsafe Component Loadings*

Taeho Kwon Zhendong Su

University of California, Davis
{kwon,su}@cs.ucdavis.edu

ABSTRACT

Dynamic loading of software components (*e.g.*, libraries or modules) is a widely used mechanism for improved system modularity and flexibility. Correct component resolution is critical for reliable and secure software execution, however, programming mistakes may lead to unintended or even malicious components to be resolved and loaded. In particular, dynamic loading can be *hijacked* by placing an arbitrary file with the specified name in a directory searched before resolving the target component. Although this issue has been known for quite some time, it was not considered serious because exploiting it requires access to the local file system on the vulnerable host. Recently such vulnerabilities started to receive considerable attention as their remote exploitation became realistic; it is now important to detect and fix these vulnerabilities.

In this paper, we present the first *automated* technique to detect vulnerable and unsafe dynamic component loadings. Our analysis has two phases: 1) apply dynamic binary instrumentation to collect runtime information on component loading (*online phase*); and 2) analyze the collected information to detect vulnerable component loadings (*offline phase*). For evaluation, we implemented our technique to detect vulnerable and unsafe DLL loadings in popular Microsoft Windows software. Our results show that unsafe DLL loading is prevalent and can lead to serious security threats. Our tool detected more than 1,700 unsafe DLL loadings in 28 widely used software and discovered serious attack vectors for remote code execution. Microsoft has opened a Microsoft Security Response Center (MSRC) case on our reported issues and is working with us and other affected software vendors to develop necessary patches.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Security

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

ISSTA'10, July 12–16, 2010, Trento, Italy.

Keywords

Unsafe Component Loading, Dynamic Analysis

1. INTRODUCTION

Dynamic loading is an important mechanism for software development. It allows an application the flexibility to dynamically link a component and use its exported functionalities. Its benefits include modularity and generic interfaces for third-party software such as plug-ins. It also helps to isolate software bugs as bug fixes of a shared library can be incorporated easily. Because of these advantages, dynamic loading is widely used in designing and implementing software.

A key step in dynamic loading is component resolution, *i.e.*, how to locate the correct component for use at runtime. Operating systems generally provide two resolution methods, either specifying the *fullpath* or the *filename* of the target component. With fullpath, operating systems simply locate the target from the given full path. With filename, operating systems resolve the target by searching a sequence of directories, determined by the runtime *directory search order*, to find the first occurrence of the component.

Although flexible, this common component resolution strategy has an inherent security problem. Since only a file name is given, unintended or even malicious files with the same file name can be resolved instead. Thus far this issue has not been adequately addressed. Operating systems may provide mechanisms to protect system resources, such as Windows Resource Protection (WRP) [2] in Microsoft Windows Vista. However, these do not prevent loading of a malicious component located in a directory searched before the directory where the intended component resides.

The problem of unsafe dynamic loading had been known for a while, but it had not been considered a serious threat because its exploitation requires local file system access on the victim host. The problem has started to receive more attention due to recently discovered remote code execution attacks. Here is an example attack scenario. An attacker sends an archive file containing a document for a vulnerable program (*e.g.*, a Word document) and a malicious DLL to a victim. If the victim opens the document, the vulnerable program will load the malicious DLL and the host machine can be subverted. Section 2.3 describes in more detail potential remote code execution attack vectors exploiting unsafe dynamic loadings.

In this paper, we present the first automated technique to detect unsafe dynamic component loadings. We cast our technique as a two-phase dynamic analysis. In the first phase, which is online, we use dynamic binary instrumentation to capture a program's sequence of events related to component loading (*dynamic profile generation*). In particular, we dynamically collect three kinds of information: 1) *system calls invoked for dynamic loading* for information on target component specifications, directory search orders, and the sequence of component loading behavior; 2) *image loading* for information on resolved component paths, and 3) *process and thread*

identifiers for multi-threaded applications. In the second phase, which is offline, we analyze the captured profile to detect unsafe component resolutions (*offline profile analysis*). We detect two types of unsafe loadings—*resolution failure* and *resolution hijacking*—for each component loading from the profile. A resolution failure corresponds to the case where the target component is not found, while a resolution hijacking corresponds to the case where there exist other directories searched before the directory containing the found target component.

To evaluate our technique, we have implemented it in a tool for detecting unsafe DLL loadings on Microsoft Windows. In the empirical evaluation, we analyzed unsafe DLL resolutions in 28 popular software applications on Microsoft Windows XP (SP3) and Microsoft Windows Vista (SP1). Our results show that unsafe DLL loadings are prevalent, and some can lead to serious security threats. More specifically, we found more than 1,700 unsafe dynamic component loadings with the administrative privilege and 19 vulnerabilities that can easily cause remote code execution. We reported these remotely exploitable vulnerabilities to Microsoft and are collaborating with Microsoft engineers to address these issues.

This paper makes the following main contributions:

- We present an effective dynamic analysis to detect vulnerable and unsafe dynamic component loadings. To our knowledge, this work introduces the first automated technique to detect and analyze vulnerabilities and errors related to dynamic component loading.
- We have realized our technique as a practical tool for detecting unsafe DLL loadings on Microsoft Windows and conducted an extensive analysis of unsafe DLL loadings on various types of popular software.
- We have discovered new remote attack vectors based on the findings from our analysis, which Microsoft has confirmed and is actively working with us and other software vendors to develop engineering solutions to patch. We also propose techniques to mitigate unsafe DLL loadings.

The remainder of this paper is structured as follows. Section 2 describes security vulnerabilities and threats in dynamic component loading, including a discussion of three types of remote code execution attacks based on unsafe dynamic loadings. In Section 3, we present our general technique to detect unsafe dynamic loadings. In Section 4, we describe background on DLL loading in Microsoft Windows and implementation details of our tool for detecting unsafe DLL loadings. Section 5 presents the evaluation of our tool, including characteristics and exploitability of the detected vulnerable and unsafe DLL loadings and our tool's performance. We also discuss techniques to mitigate unsafe DLL loadings and generality of our proposed approach (Section 6). Finally, we survey related work (Section 7) and conclude with a discussion of future work (Section 8).

2. UNSAFE COMPONENT LOADING

This section describes dynamic loading of components, types of unsafe loadings, and remote attack vectors for vulnerable dynamic loading.

2.1 Dynamic Loading of Components

Software components often utilize functionalities exported by other components such as shared libraries at runtime. This operation is generally composed of three phases: *resolution*, *loading*, and *usage*. Specifically, an application resolves the needed target components, loads them, and utilizes the desired functions provided by them.

Component inter-operation can be achieved through *dynamic loading* provided by operating systems or runtime environments.

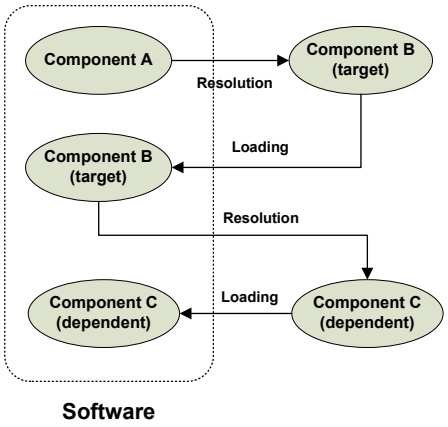

Figure 1: Dynamic loading procedure.

For example, the `LoadLibrary` and `dlopen` system calls are used for dynamic loading on Microsoft Windows and Unix-like operating systems respectively. Dynamic loading is generally done in two steps: *component resolution* and *chained component loading*.

Component resolution To resolve a target component, it is necessary to specify it correctly. To this end, operating systems provide two types of target component specifications: *fullpath* and *filename*. For fullpath specification, operating systems resolve a target component based on the provided fullpath. For example, a fullpath specification `/lib/libc-2.7.so` for the libc library in Linux determines the target component using the specified full path. For filename specification, operating systems obtain the full path of the target component from the provided file name and a dynamically determined sequence of search directories. In particular, an operating system iterates through the directories until it finds a file with the specified file name, which will be the resolved component. For example, suppose that a target component is specified as `midimap.dll` and the directory search order is given as `C:\Program Files\iTunes;C:\Windows\System32; ...;$PATH` on Windows. If the first directory containing a file with the name `midimap.dll` is `C:\Windows\System32`, the resolved full path is determined by this directory.

Chained component loading In dynamic loading, the full path of the target component is determined by its specification through the resolution process, and the component is incorporated into the host software if it is not already loaded. During the process of incorporating the target component, the component's load-time dependent components are also loaded. Figure 1 illustrates the general procedure of dynamic loading. Suppose component B is loaded by component A. B's dependent components (*e.g.*, component C) are also loaded. We can usually obtain information on B's dependent components from B's file description. This process of chained component loading is repeated until all dependent components have been loaded.

2.2 Unsafe Component Resolution

Although dynamic loading is a critical step in software execution, it also has an inherent security implication. Specifically, a loaded target component is only determined by the specified file name. This can lead to the loading of unintended or even malicious components and thus may allow arbitrary code execution. For example, an attacker can trick a vulnerable web browser to resolve a spyware file with the specified file name instead of the intended component.

To mitigate this problem, operating systems provide functionalities to prevent certain components from being replaced. For example, Windows Vista has Windows Resource Protection (WRP) [2] to protect system resources, and Unix-like operating systems require root privilege for modifying system files. However, these protections are

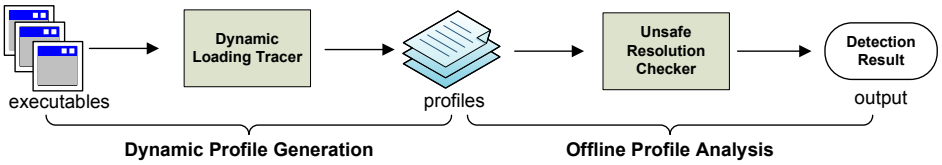

Figure 2: Analysis process.

not sufficient to prevent unsafe component loadings. Due to defects in the resolution process, it is possible to hijack the loading of an intended component. We classify two types of unsafe component resolution: *resolution failure* and *resolution hijacking*.

Resolution failure This occurs when an application fails to resolve a target component because the file does not exist in the specified path or the specified search directories. If this happens, any file with the same specified name in the directories searched by the application can be loaded, and attackers can execute arbitrary code by placing malicious component files in such directories.

Resolution hijacking Although the correct component is resolved, there may still exist a vulnerability that makes hijacking possible. In particular, if the resolution of a component satisfies the following conditions, it may still be possible to hijack the loaded component:

1. The target component is specified by its file name;
2. The resolution is determined by iteratively searching a sequence of directories; and
3. There exists another directory searched before the one containing the target component.

If a dynamic loading satisfies the above conditions, an attacker can attempt to place a malicious component with the same specified name in such an earlier searched directory. When the OS tries to resolve the target component, the malicious component is found first and loaded in place of the intended component. In such a case, arbitrary code execution becomes possible.

2.3 Dynamic Loading-related Remote Attacks

As we mentioned in Section 2.2, unsafe component resolutions may cause an application to load unintended components. This issue had been known for a long time, but it had not been considered a serious threat because it requires local file system access on the victim host for exploitation. Recently, realistic attacks exploiting vulnerable component loading have been discovered, including ones by us. In this section, we describe these attack vectors.

2.3.1 "Carpet Bomb"-based attack

The *Carpet Bomb* attack [1] can lead to remote code execution in conjunction with unsafe DLL loading on Microsoft Windows. In particular, when the Safari browser accesses a malicious web page, attackers can make the browser automatically download arbitrary files to the user's Desktop directory without any prompting. This is referred to as the Carpet Bomb attack. This flaw leads to remote code execution if a vulnerable application checks in the Desktop directory first for resolving a DLL. For example, suppose `sqmapi.dll` is downloaded onto the victim's Desktop directory through the Carpet Bomb attack. When Internet Explorer 7 runs, it loads this DLL file and executes arbitrary code [19]. Microsoft released software patches [25, 26] to fix this vulnerability.

2.3.2 "Shortcut with Component" attack

Sending a victim an archive file containing a shortcut to a vulnerable program and a malicious component can also cause remote code execution. If the vulnerable program starts up via the shortcut, it loads the component and executes malicious code.

This flaw can be exploited through social engineering-based attacks. For example, Opera 9.64 running via its shortcut will load `aspell-15.dll` placed in the same directory as the shortcut. Attackers can deceive the victim to run Opera through its shortcut to access interesting web sites such as Facebook. This way they can exploit this vulnerability by making the browser load the provided malicious DLL.

Furthermore, this attack vector can be combined with the Carpet Bomb attack. Because shortcuts tend to be placed in the Desktop directory, running a vulnerable application such as Opera via its shortcut can load the relevant components stored on the Desktop through the Carpet Bomb attack.

2.3.3 "Document with Component" attack

Opening a document can load particular files placed in the same directory as the document. This vulnerability can be exploited to launch remote code execution attacks by sending a victim an archive file containing a document and a malicious component.

For example, opening a Microsoft Word document can load the `GoogleDesktopCommon.dll` file if Google Desktop is installed on the victim host. In particular, a third-party component, `Google Desktop Office Addin`, is registered in Microsoft Word 2007 when Google Desktop is installed. The component is loaded when the programs execute, but the document directory is searched before resolving the DLL during loading of the component. This flaw can lead to serious security threats in Microsoft Word 2007.

3. DETECTION OF UNSAFE LOADINGS

As we mentioned in Section 2.3, unsafe component loading can cause serious security vulnerabilities in software. In this section, we present a dynamic analysis technique to detect unsafe resolutions.

Figure 2 shows the high-level overview of our analysis process, which is composed of two phases: *dynamic profile generation* and *offline profile analysis*. To detect unsafe component resolutions, we first capture a sequence of system-level actions for dynamic loading during a program's execution. We use dynamic binary instrumentation to generate its runtime profile. We then reconstruct dynamic loading information from the profile offline and check safety conditions for each resolution. Because our technique only requires binary executables, it is robust and can be applied to analyze not only open source applications but also commercial off-the-shelf products,

Alternatively, we could also detect unsafe component loading during the program execution. However, we divide our analysis into two phases (*i.e.*, the dynamic profile generation and the offline profile analysis) to reduce the performance overhead incurred during dynamic binary instrumentation.

3.1 Dynamic Profile Generation

In this phase, we instrument runtime executions of the binary executable under analysis to capture a sequence of system-level actions for dynamic loading of components. In particular, we collect three types of information during the instrumented program execution: *system calls invoked for dynamic loading*, *image loading*, and *process and thread identifiers*. The collected information is stored as a profile for the instrumented application and will be analyzed in the offline profile analysis phase.

Type	Conditions
Resolution failure	1. Target component is not found
Resolution hijacking	1. Target component is specified by its name 2. Target component is resolved by iterating through multiple directories

Table 1: Conditions for detecting unsafe component loadings.

System calls invoked for dynamic loading System call analysis is a widely used analysis technique to understand program behavior because a sequence of invoked system calls (with names of the invoked functions and their arguments) can provide useful information on program execution. To capture system-level actions for dynamic component loading, we instrument system calls that cover all possible control-flow paths of the dynamic loading procedure, which enables us to reconstruct the procedure offline.

Besides the name of an instrumented system call, we also collect its parameter information for detecting unsafe component resolutions. Specifically, the target component specification and the directory search order can be obtained from the system call parameters. Although the directory search order can vary according to the underlying system and program setting, it is computed by operating systems at the higher-level and provided as parameters to the relevant system calls for dynamic loading. Furthermore, results of the instrumented system calls provide both the control flow in the loading procedure and error messages generated by the operating systems. Such information is used for the reconstruction of the dynamic loading procedure and the detection of unsafe resolutions.

Image loadings We also capture actual loadings of target components via dynamic binary instrumentation. The loading information is needed for reconstructing the loading procedure in combination with the information captured by system call instrumentation. It also indicates the resolved full path determined by the loading procedure. We use this resolved path to detect resolution hijacking.

Process and thread identifiers Because our approach is based on system call instrumentation, it is important to consider multi-threaded applications. If the target program uses multi-threads and each thread loads a component dynamically, the instrumented system calls for each loading can be interleaved, which makes it difficult to correctly reconstruct the loading procedure of each thread. To solve this problem, we capture process and thread identifiers along with the other information on instrumented system calls. With this additional information, we can analyze dynamic loadings of each thread by grouping its system calls using these recorded identifiers.

3.2 Offline Profile Analysis

In this phase, we extract each component loading from the profile and detect defects in the resolution of a target component and its dependent components (*cf.* Section 2).

In the first step of this offline phase, we extract each component loading from the profile. To this end, we first group a sequence of actions in the profile by process and thread identifiers as the actions performed by different threads may be interleaved due to context switching. This grouping separates the sequences of dynamic loadings performed by different threads. Next, we divide the sequence for each thread into sub-sequences of actions, one for each distinct dynamic loading. This can be achieved by using the first invoked system call for dynamic loading (*e.g.*, dlopen) as a delimiter. After this step, we obtain a list of groups, each of which contains a sequence of actions for loading a component at runtime. This gives the possible control-flows in the dynamic loading procedure. Note that each group contains loading actions for both the target component and the load-time dependent components (*cf.* Section 2.1).

Our analysis detects the two types of unsafe component resolution that we discussed in Section 2.2: resolution failure and resolution hijacking. To this end, we check conditions in Table 1, which are directly derived from the definition of each unsafe component resolution, for each component loading. Details of our offline profile analysis are given in Algorithm 1.

Algorithm 1 OfflineProfileAnalysis

```
Input: S (a sequence of actions for a dynamic loading)
Auxiliary functions:
  TargetSpec(S): return target specification of S
  DirSearchOrder(S): return directory search order used in S
  ImgLoad(S): return the image loadings in S
  ResolutionFailure(S): return the resolution failures in S
  ChainedLoading(S): return actions for the chained loadings in S
  IsResolutionHijacking(filename, resolved_path, search_dirs):
    check whether resolution hijacking is possible

1: img_loads ← ImgLoad(S)
2: failed_resolutions ← ResolutionFailure(S)
3: if |img_loads| == 0 then
4:     if |failed_resolutions| == 1 then
5:         Report this loading as a resolution failure
6:     end if
7: else
8:     spec ← TargetSpec(S)
9:     dirs ← DirSearchOrder(S)
10:    if spec is the filename specification then
11:        resolved_path ← img_loads[0].resolved_path
           // retrieve the first load
12:        if IsResolutionHijacking(spec,resolved_path,dirs) then
13:            Report this loading as a resolution hijacking
14:        end if
15:    end if
16:    chained_loads ← ChainedLoading(S)
17:    for each_load in chained_loads do
18:        OfflineProfileAnalysis (each_load)
19:    end for
20: end if
```

Resolution failure of target component To detect failed resolution of a target component, we simply check the number of image loads and failed resolutions during the dynamic loading procedure. In particular, if no image is loaded and its resolution is failed, we report it as a resolution failure (lines 3–6).

Resolution hijacking of target component Lines 10–15 describe how to detect resolution hijacking of a target component. We first check whether the target component is specified by its file name, because a full path specification does not iterate through the search directories for resolution. If the file name is used, we retrieve the resolved path of the target component by retrieving the first element of a list of image loads in the dynamic loading procedure. Note that the first element of the list corresponds to the target component, because 1) there exists no image load in the loading procedure if the target component is already loaded or its resolution fails, and 2) the target component is always loaded for the first time during its runtime loading. Based on the resolved full path, the target component specification and the applied directory search order, we determine whether to classify this as a resolution hijacking by checking the directories searched before the resolution.

Unsafe component resolution by chained loadings In lines 16–19, we detect unsafe component resolutions in the chained loading procedure by performing the offline profile analysis recursively. In particular, we extract each component loading from the chained loadings and recursively apply our earlier described techniques to detect unsafe resolutions of the component.

4. IMPLEMENTATION

To evaluate our proposed technique, we have developed a tool to detect unsafe DLL loadings on Microsoft Windows. In this section, we explain background on the DLL loading procedure in Windows, discuss implementation details of our tool, and show examples of generated profiles and unsafe DLL loadings.

Search Type	Order
Standard	1. The directory of the application loaded 2. The system directory 3. The 16-bit system directory 4. The Windows directory 5. The current directory 6. The `PATH` environment variable
Alternate	1. The directory specified by *lpFileName* 2. The system directory 3. The 16-bit system directory 4. The Windows directory 5. The current directory 6. The `PATH` environment variable
SetDllDirectory -based	1. The directory of the application loaded 2. The directory specified by *lpPathName* 3. The system directory 4. The 16-bit system directory 5. The Windows directory 6. The `PATH` environment variable

Table 2: DLL search orders of SafeDllSearch mode.

4.1 Background on DLL Loading

4.1.1 Target DLL resolution

Microsoft Windows supports the two aforementioned types of target DLL specifications: fullpath and filename. For the filename specification, there exist Windows-specific mechanisms to resolve target DLLs. In particular, Microsoft Windows supports *Side-by-Side Assembly* [32] and maintains *Known DLLs* to determine the target DLL fullpath directory without the directory iteration.

Side-by-Side Assembly This technique has been provided to mitigate DLL Hell [33]. In this technique, Windows stores multiple versions of a DLL in the WinSxS directory and loads the desired DLL on demand. For example, when Microsoft Word 2007 loads the Microsoft C runtime library by using its file name (*i.e.*, `MSVCR80.dll`), its full path is determined by a sub-directory of the Windows SxS directory (*i.e.*, `C:\WINDOWS\WinSxS\...\MSVCR80.dll`) without iteratively searching a list of directories. In general, the full path is determined by the existence of its corresponding `Manifest`, an XML document which is usually embedded in the executable. More details can be found in an MSDN article [32].

Known DLLs The Microsoft Windows operating systems maintain a set of *known DLLs* that correspond to core system DLLs and their load-time dependent ones. The set of core DLLs is determined by the registry key `HKLM\System\CurrentControlSet\Control\Session Manager\KnownDLLs`. If the target DLL is among the known DLLs, its full path is resolved by the directory specified in the `DllDirectory` value located in the registry, which is `%SystemRoot%\system32` by default on 32-bit Windows XP and Vista.

4.1.2 Directory search order

As we mentioned in Section 2, dynamic component resolution based on filename requires a directory search order, which is determined by system and program settings at runtime. According to MSDN [12], the `SafeDllSearchMode` registry key, the `LOAD_WITH_ALTERED_SEARCH_PATH` flag, and the `SetDllDirectory` system call determine five possible types of directory search orders at runtime, which are *standard search order (SafeDllSearchMode)*, *alternate search order (SafeDllSearchMode)* and *SetDllDirectory-based SearchOrder*. Table 2 shows the search orders when SafeDllSearchMode is enabled.

Standard Search Order The standard search order is the default directory search order in Microsoft Windows, which has two types determined by whether SafeDllSearchMode is enabled. The SafeDllSearchMode was introduced by Microsoft Windows 2000 SP4, and it has been enabled by default since Microsoft Windows XP SP2. For the Standard Search Order of the SafeDllSearchMode, there exist six types of directories to search for DLL resolution (see Table 2). If the SafeDllSearchMode is disabled, the priority of the current directory is elevated to the second one.

Alternate Search Order The standard search order can be modified by invoking the system call `LoadLibraryEx` with the flag `LOAD_WITH_ALTERED_SEARCH_PATH`. Similar to the standard search order, the SafeDLLSearchMode can be applied to the alternate search order. However, it places the directory of the loading DLL to the first directory to search; the target DLL is specified by its full path, which corresponds to an *lpFileName* parameter of the LoadLibraryEx function.

SetDllDirectory-based Search Order Microsoft has provided the `SetDllDirectory` system call to enable developers to manipulate the search order since Microsoft Windows XP SP1. The SetDllDirectory function makes it possible to replace the current directory with an arbitrary directory specified by an *lpPathName* parameter. Also, the current directory can be removed from the search order by invoking the system call with the empty string as the parameter. Note that this search order is independent from the SafeDllSearchMode; the search order is determined as shown in Table 2 regardless of the SafeDllSearchMode.

4.1.3 Chained DLL loading

According to Microsoft [3], there exist two types of load-time dependencies among DLLs: implicit dependency and forwarded dependency.

Implicit Dependency If a DLL A and a DLL B are linked at compile/link time, and the source code of DLL A calls one or more functions exported from DLL B, DLL A has *implicit dependency* on DLL B. Note that implicit-dependent DLLs are determined by function calls invoked by the source code of the loading DLL. Even though the function is not invoked at runtime, the DLL exporting the function is also loaded. The loading DLL's `Import Directory Table`, one entity of the PE executable file format [24], contains its implicit-dependent DLLs.

Forwarded Dependency While this dependency is similar to implicit dependency, it differs in what the DLL that implements the invoked functions is. For the load-time dependency, the functions a loading DLL invokes are directly implemented in its dependent DLLs. However, for forwarded dependency, the implementation of the invoked function call simply forwards control to the actual code implemented in the other DLL. In this case, the loading DLL has *forwarded dependency* on the DLL containing the forwarded implementation. For example, the `GetLastError` function of `Kernel32.DLL` is forwarded to the `RtlGetLastWin32Error` function of `ntdll.dll`.

4.2 Implementation Details

In order to generate profiles for DLL loading behaviors, we utilize *Pin* [23], an open source dynamic binary instrumentation tool. We record a sequence of information on the system calls of interest, image loading, and process/thread identifiers with functions provided by the tool.

As we mentioned in Section 3, the system calls to instrument are determined to cover all possible control-flow paths in the DLL loading procedure. Because this information is not well-documented, we reverse-engineered the `LdrLoadDll` function of `ntdll.dll` using the IDA Pro Disassembler [18] based on detailed analysis of DLL loadings for Windows 2000 [38]. Table 3 describes the system calls instrumented in our implementation. We chose the name of each system call based on the analysis of the disassembler, which uses the Windows symbol package.

Instrumenting a system call requires information where it is located in the address space such as its starting virtual address. However, this information is difficult to obtain reliably because DLLs follow the PE format and can be relocated in the address space. Also, Address Space Layout Randomization [39] is one of the de-

```
1  (1744,1050) LdrLoadDll midimap.dll C:\Program Files\iTunes;C:\WINDOWS\system32;
2                                     C:\WINDOWS\system;C:\WINDOWS;.;$PATH
3  (1744,1050) RtlDosApplyFileIsolationRedirection_Ustr midimap.dll c0150008
4  (1744,1050) LdrpLoadDll midimap.dll
5  (1744,1050) LdrpCheckForLoadedDll midimap.dll   0
6  (1744,1050) LdrpMapDll midimap.dll
7  (1744,1050) LdrCheckForKnownDlls midimap.dll    0
8  (1744,1050) LdrpResolveDllName midimap.dll      0
9  (1744,1050) IMG_LOAD C:\WINDOWS\system32\midimap.dll 77b80000
10 (1744,1050) LdrpMapDll_RET midimap.dll
11 (1744,1050) LdrpWalkImportDescriptor 274658
12 (1744,1050) LdrpLoadImportModule msvcrt.dll
13 (1744,1050) RtlDosApplyFileIsolationRedirection_Ustr msvcrt.dll c0150008
14 (1744,1050) LdrpCheckForLoadedDll msvcrt.dll    1
15                ...
16 (1744,1050) LdrpLoadImportModule WINMM.dll
17 (1744,1050) RtlDosApplyFileIsolationRedirection_Ustr WINMM.dll c0150008
18 (1744,1050) LdrpCheckForLoadedDll WINMM.dll 1
19 (1744,1050) LdrpWalkImportDescriptor_RET 274658
20 (1744,1050) LdrLoadDll_RET midimap.dll
```

Figure 3: Profile excerpt of iTunes.

```
1  (ac0,114c) LdrLoadDll GoogleDesktopCommon.dll C:\Program Files\Microsoft Office\Office12;
2                                                 C:\WINDOWS\system32;C:\WINDOWS\system;
3                                                 C:\WINDOWS;.;$PATH
4  (ac0,114c) RtlDosApplyFileIsolationRedirection_Ustr GoogleDesktopCommon.dll c0150008
5  (ac0,114c) LdrpLoadDll GoogleDesktopCommon.dll
6  (ac0,114c) LdrpCheckForLoadedDll GoogleDesktopCommon.dll 0
7  (ac0,114c) LdrpMapDll GoogleDesktopCommon.dll
8  (ac0,114c) LdrCheckForKnownDlls GoogleDesktopCommon.dll 0
9  (ac0,114c) LdrpResolveDllName GoogleDesktopCommon.dll   c0000135
10 (ac0,114c) LdrpMapDll_RET GoogleDesktopCommon.dll
11 (ac0,114c) LdrLoadDll_RET GoogleDesktopCommon.dll
```

Figure 4: A resolution failure in Microsoft Word 2007.

Name	Description
LdrLoadDll	Load a DLL
LdrpLoadDll	Load a DLL (private function)
RtlDosApplyFileIsolationRedirection_Ustr	Apply the redirection of the DLL specification
LdrpCheckForLoadedDll	Check whether or not a target DLL was loaded before
LdrpCheckForKnownDlls	Check whether or not a target DLL is one of known DLLs
LdrpResolveDllName	Resolve the fullpath of the target DLL specification
LdrpMapDll	Map a DLL to the address space
LdrpWalkImportDescriptor	Load loadtime-linking DLLs of the target DLL
LdrpLoadImportModule	Load a loadtime-linking DLL

Table 3: Instrumented system calls.

fault configurations of Windows Vista, which can randomize the base addresses of the loading images to mitigate memory corruption attacks. To address this problem, we identify target virtual addresses to instrument by combining the last two bytes of the address with consecutive bytes of the predefined number starting from the virtual address. For example, the `LdrLoadDll` of `ntdll.dll` of version 5.1.2600.5755 is located at 0x7C9463C3, and it can be identified by using its last two bytes (*i.e.*, 0x63C3) and the consecutive eight bytes starting from that address (*i.e.*, 0x6518680000026C68) as the signature for the address. The eight bytes are enough not to induce any false positives in our test environment, and we were able to reliably instrument these system calls.

To facilitate the offline profile analysis for reconstructing dynamic loadings, we also capture the execution of `return` instructions of particular system calls (*i.e.*, `LdrLoadDll_RET`, `LdrpMapDll_RET`, and `LdrpWalkImportDescriptor_RET`). The system calls and their corresponding `return` instructions can be used to determine the scope of component loadings in the profile.

To implement our offline profile analysis, we wrote a Python script (239 lines) to extract each DLL loading from the profile and detect unsafe DLL loadings.

4.3 DLL-loading Behavior Profile Example

Figure 3 shows part of a generated profile, which describes runtime loading procedure of `midimap.dll` in `iTunes`. The profile is composed of two parts. Lines 1–10 represent runtime loading of `midimap.dll`, and lines 11–20 correspond to loadings of its loadtime dependent DLLs such as `msvcrt.dll` and `WINMM.dll`. The profile provides detailed information on DLL loading. The first and second items in each line describe the process/thread identifier and the loading behavior represented by the corresponding system call name or a tag for image loading, `IMG_LOAD`, respectively. According to the type of the loading behavior, each line contains different information required for the analysis: 1) lines 1–2 contain the target DLL specification given by its file name and the directory search order to be applied for the current DLL loading; 2) lines 3, 5, and 7 show information on the DLL redirection and the checks for the known DLL and the loaded DLL by the return value of the corresponding system calls, respectively; 3) line 8 shows result of the DLL name resolution; 4) line 9 shows the resolved DLL path for the given DLL specification; and 5) lines 11 and 19 give the scope of the behaviors performed for the chained loading due to the loaded target DLL. Based on this information stored in the profiles, we perform offline analysis to detect unsafe DLL loadings.

4.4 Example Unsafe DLL Loading

We describe how our detection technique works by showing examples for each type of unsafe DLL loadings.

Resolution Failure Figure 4 shows a resolution failure type in Microsoft Word 2007. The application tries to resolve a DLL speci-

Software	XP				Vista			
	Resolution Failure		Resolution Hijacking		Resolution Failure		Resolution Hijacking	
	Target	Chained	Target	Chained	Target	Chained	Target	Chained
MS Office								
Access 2007	0/0	0/0	0/7	0/9	0/0	0/0	0/17	0/5
Excel 2007	0/1	0/0	0/7	0/7	0/1	0/0	0/12	0/5
Word 2007	1/2	0/0	0/16	0/9	1/2	12/0	0/20	0/26
PowerPoint 2007	1/2	0/0	0/14	0/9	1/2	0/0	0/12	0/16
Outlook 2007	1/1	0/0	0/9	0/12	1/1	0/0	0/11	0/10
Visio 2007	2/0	0/0	0/8	0/9	2/0	0/0	0/6	0/4
Onenote 2007	0/0	0/0	0/8	0/6	0/0	0/0	0/8	0/7
Web Browser								
Internet Explorer 8	0/0	0/0	0/16	0/18	0/1	0/0	0/18	0/18
Firefox 3.0	3/1	1/0	0/5	0/12	3/1	1/0	0/12	0/20
Chrome 2.0	0/0	0/0	0/13	0/16	0/0	0/0	0/10	0/13
Opera 9.64	0/2	0/0	0/2	0/9	0/2	0/0	0/10	0/20
Safari 4.0	0/1	0/0	0/34	0/18	0/0	0/0	0/9	0/5
PDF Reader								
Acrobat Reader 9.1.2	0/0	0/0	0/6	0/5	0/0	0/0	0/11	0/11
Foxit Reader 3.0	0/0	0/0	0/3	0/3	0/0	0/0	0/6	0/3
Messenger								
Windows Live Messenger 2009	0/0	0/0	0/3	0/4	0/0	0/0	0/22	0/12
Pidgin 2.5.8	1/0	0/2	0/25	0/9	1/0	0/1	0/11	0/37
Google Talk Beta	0/1	0/0	0/10	0/20	0/1	0/0	0/19	0/12
Yahoo! Messenger 9.0	0/1	0/0	0/16	0/24	0/1	0/0	0/24	0/21
Skype 3.0	0/0	0/0	0/13	0/31	0/1	0/0	0/28	0/19
Image Viewer								
Picasa 3	0/0	0/0	0/9	0/18	0/0	0/0	0/14	0/13
Irfan View 4.25	0/0	2/0	0/10	0/6	0/0	0/0	0/17	0/17
Multimedia Player								
Itunes 8.2.1	0/1	0/0	0/34	0/31	0/2	0/0	0/25	0/21
Winamp 5.56	2/1	0/1	0/6	0/13	3/0	0/0	0/21	0/25
Realplayer 10.0	0/2	0/0	0/20	0/21	2/3	0/0	0/27	0/35
Windows Media Player 11	0/1	0/1	0/19	0/27	0/1	0/1	0/34	0/37
QuickTime 7.6.2	0/0	0/0	0/18	0/23	0/1	0/0	0/25	0/32
Others								
Google Desktop 5.8	0/0	0/0	0/8	0/12	0/0	0/0	0/14	0/5
Google Earth 5.0	1/3	0/0	0/12	0/15	1/4	0/0	0/19	0/16

Table 4: Prevalence of unsafe DLL loadings (fullpath/filename).

fied by `GoogleDesktopCommon.dll`. However, the resolution fails because there does not exist a file in the directories determined by the applied directory search order. `LdrpResolveDllName` on line 9 is unable to locate the DLL, which corresponds to the error message `c0000135`.

Resolution Hijacking Figure 3 is an example of resolution hijacking. The target DLL specified by `midimap.dll` is resolved to `C:\Windows\System32\midimap.dll` by checking the file of the specified name located in the directories based on the directory search order. Because the system directory is the second directory to be searched by the OS, placing an arbitrary file of the specified name in the first searched directory (*i.e.*, `C:\Program Files\iTunes\midimap.dll`) can lead to the hijacking of the intended DLL loading.

5. EVALUATION

In this section, we analyze the unsafe DLL loadings detected from the popular software on Windows XP Professional SP3 and Windows Vista Home Edition SP1, evaluate their severity by showing exploitability of the detected results by local and remote attacks, and measure performance of our technique.

5.1 Analysis of Unsafe DLL Loadings

In our evaluation, the detection of unsafe DLL loadings is performed with the administrator privilege because 1) we aim at detecting all possible unsafe DLL loadings to evaluate the worst case, and 2) most Windows users have the administrator privilege [34], in contrast to Unix/Linux-based operating systems.

Table 4 shows how prevalent unsafe DLL loadings are for each phase of dynamic loading (*i.e.*, target and chained component loading). We include detection results for a few different types of software developed by major software vendors. The two numbers in each entry in the table represent the numbers of unsafe DLL resolutions caused by fullpath and filename specifications, respectively. According to the table, unsafe DLL loadings are common programming mistakes in developing these applications. We found more than 1,700 instances of unsafe dynamic loadings, 786 under XP and 982 under Vista. Considering the types of these unsafe DLL loadings, resolution hijacking is largely responsible for them. In particular, resolution hijackings in Windows XP and Vista correspond to 95% (747/786) and 94% (927/982) of the total unsafe loadings, respectively. In the following subsections, we give a detailed analysis of each type of the unsafe DLL loadings.

Type	FULLPATH		FILENAME	
	XP	Vista	XP	Vista
Application DLL	3	4	10	8
Third-party component DLL	7	21	9	9
Language support DLL	5	3	0	0
Unsupported DLL	0	0	5	10

Table 5: Analysis of resolution failures.

5.1.1 Resolution failures

Table 5 shows types of target DLLs whose resolutions fail. In particular, for the fullpath and filename specifications, there exist four types of target DLLs: application DLL, third-party component DLL, language support DLL, and unsupported system DLL.

Application DLL Many applications do not include application-specific DLLs in their releases, which can cause resolution failures of these libraries. For example, `Google Earth 5.0` tries to load

DLL-hijacking Directory	XP		Vista	
	Runtime	Loadtime	Runtime	Loadtime
Application directory	396	282	462	392
Application library directory	9	9	0	20
System directory	2	0	4	0
Current directory	1	0	1	0
Part of $PATH	1	0	1	0
Plug-in directory	0	6	0	7
WBEM directory	0	10	0	11
Driver directory	0	14	0	16
System-hook source directory	0	31	1	36

Table 6: Types of DLL-hijacking directories.

Resolved Directory	XP	Vista
System directory	727	905
Application library directory	18	18
Application plug-in directory	1	3
Application directory	1	1

Table 7: Types of resolved directories.

`collada.dll` in the application directory when it starts up, but such a library is not included in the release.

Third-party component DLL Third-party components embedded in applications can also cause DLL resolution failures. There are two main reasons for this: 1) difference of the directory search order between the application and the component, and 2) the loadings of missing DLLs by the components.

For the first reason, when an application loads a third-party component, the applied directory search order for the resolution is determined by the setting of the running application. Because the intended directory search order for the component can be different from the applied one, the DLL resolution by the component can fail. Figure 4 describes an example of resolution failure because of attempting to load a third-party component. In particular, `Google Desktop` registers a `Google Desktop Office Addin` to Microsoft Word and PowerPoint, and it is loaded when these applications run. During the loading procedure, the component tries to load `GoogleDesktopCommon.dll`, which is located in the directory of `Google Desktop`. However, because the applied directory search order does not contain this directory, this resolution fails.

Similar to resolution failures of an application, third-party components may attempt to load DLLs that do not exist on the system due to careless programming. For example, Microsoft Word and PowerPoint 2007 try to load driver files of the printers installed on the system during their startup. However, some HP printer drivers try to load the non-existing `HPProfiler.dll` during the driver loading process, which causes the resolution failure.

Language-support DLL Many applications load resource files for language-support, but these files may not exist on the system. For example, Microsoft Visio 2007 loads `VISIOKOR.DLL` in its application directory when it runs on the Korean version of Microsoft Windows XP Professional SP3. However, the release of Visio does not contain such a file.

Unsupported system DLL Windows Vista provides some DLL files to support new features. Because these DLLs do not exist on Windows XP, it is necessary to consider the version of the current operating system when loading these files. Otherwise, it can cause resolution failures. However, many applications developed for both versions of Windows usually do not consider this issue. For example, `Winamp 5.56` loads `DWMAPI.dll`, a DLL for Windows Manager API in Windows Vista, even if the current operating system is Windows XP.

In Windows Vista Home Edition SP1, many applications accessing the web try to load a Novell NetIdentity HTTP Filter `rpawinet.dll`. However, the file is not installed in the system by default, which causes a resolution failure.

5.1.2 Resolution hijackings

Tables 6 and 7 show distributions of types of DLL-hijacking and resolved directories. These results indicate that most unsafe resolutions of system DLLs can be hijacked from the directories of the applications loading them.

Table 6 also shows that there exist types of DLL-hijacking directories that are not related to the application such as the plug-in directory. This is because the target DLL is specified by its full path, and the alternate search order in Table 2 is applied to load its load-time-dependent DLLs, which searches the directory of the target DLL first. For example, Yahoo! Messenger 9.0 loads `C:\Windows\System32\WBEM\FASTPROX.DLL` to use a WMI feature [40] of Microsoft Windows based on the alternate search order. After loading the `FASTPROX.DLL`, its load-time dependent DLLs such as `MSVCP60.DLL` are resolved to the file in the system directory. In this case, the WBEM directory can serve as a DLL-hijacking directory, because the directory is searched before the resolution of the target DLL based on the applied search order.

5.2 Severity

In this section, we evaluate exploitability of unsafe component loadings in terms of local and remote attacks. Local attacks assume that attackers can access the local file system on a victim host, while remote attacks assume that attackers can only send data to the victim user.

5.2.1 Local attacks

As we mentioned in Section 2, unsafe DLL loading can be performed by placing a file with the specified name in the DLL-hijacking directories. To exploit this security vulnerability for local attacks, attackers require write permission to the DLL-hijacking directory. According to Table 5 and Table 6, most of the directories are not writable by non-admin users. Therefore, if attackers do not have administrator privilege, most local attacks can be prevented. However, according to Microsoft [34], most Windows users run with administrative privilege. Because of this fact, unsafe DLL loadings should still be considered serious security issues.

5.2.2 Remote attacks

To accomplish remote attacks exploiting unsafe component loadings, attackers need to place malicious files in the DLL-hijacking directories from remote sites. However, accessing the file system of a remote host is generally prohibited. For example, the system directory is not accessible remotely unless the directory is shared to the remote user or the system is exploited by other vulnerabilities to enable this. Because of the difficulty in remote exploitation, unsafe component loadings have not been considered serious security threats. However, as we mentioned in Section 2.3, several remote attack vectors based on unsafe component loading have been recently discovered.

To find remote attacks on Microsoft Windows, we focus on unsafe DLL loadings caused by the following three conditions: *resolution failure*, *filename specification*, and *standard or alternate search order*. According to the directory search orders discussed in Table 2, this type of unsafe DLL loading makes OS check the current direc-

OS	Software	DLL name	DLL-loading time	Precondition
XP/Vista	iTunes 8.2.1.6	ipodvoiceover.dll	On execution	
	Opera 9.64	aspell-15.dll	On execution	
		GoogleDesktopCommon.dll	On execution	Google Desktop installation
	RealPlaer 10.5	RIO300.dll or RIO500.dll	On termination	
Vista	iTunes 8.2.1.6	rpawinet.dll	On execution	
	RealPlayer 10.5	rpawinet.dll	On execution	
	Quick Time Player 7.6.2	rpawinet.dll	On update check	

Table 8: "Shortcut with component" attacks.

OS	Software	DLL name	DLL-loading time	Precondition
XP/Vista	MS Word/PowerPoint 2007	HPProfiler.dll	On document open	HP printer driver installation
		GoogleDesktopCommon.dll	On document open	Google Desktop installation
Vista	Foxit Reader 3.0	rpawinet.dll	On update check	

Table 9: "Document with component" attacks.

OS	Software	DLL name	DLL-loading time
Vista	Internet Explorer 8	rpawinet.dll	On execution

Table 10: A threat combined with "Carpet Bomb" attack.

tory corresponding to "." during DLL resolution. In this case, the directory may be writable from the remote site because of software bugs. The blended threat combined with the Safari's Carpet Bomb attack discussed in Section 2.3.1 exploits this flaw. In particular, when Internet Explorer 7 tries to resolve the target DLL, the current directory is checked before the resolution and corresponds to the Desktop directory. This makes the program load and execute malicious DLL files on the Desktop directory, which are downloaded through the Carpet Bomb attack.

Based on this observation, we detect potential remote attacks by checking whether or not the current directory is writable by remote users when each resolution failure based on the filename specification happens. In this evaluation, we consider the following two types as remotely writable directories: *directory sent by remote users* and the *Desktop directory*. For the first directory type, attackers can send arbitrary directory structures by using archive files similar to malware propagation via e-mail. Considering the Desktop directory, we assume that the Carpet Bomb attack is possible.

Using this technique, we detect remote code execution vulnerabilities from resolution failures for the software listed in Table 5. Based on our detection result, we discover three types of remote attack vectors discussed in Section 2.3.

Shortcut with component The current directory of applications run via their shortcuts may be the same directory as the shortcuts at the point of the resolution failure. In this case, the shortcut directory can serve as the DLL-hijacking directory for remote code execution. Table 8 shows information on the discovered remote attacks of this category. For example, Quick Time Player 7.6.2 run via its shortcut on Windows Vista has a flaw where it loads `rpawinet.dll` located in the same directory as the shortcut on its update check. This vulnerability can lead to remote code execution attacks because the program generally checks its update on its execution. Note that this type of attack can be combined with the Carpet Bomb attack because the usual location of the shortcut is the desktop directory.

One interesting discovery is the attack caused by third-party component loading. For example, when Opera 9.64 runs via its shortcut on the host where Google Desktop is installed, it loads `GoogleDesktopCommon.dll` on its startup. This vulnerability shows that third-party components can cause software hosting them to perform unsafe component loading, which can be exploited by attackers for remote code execution.

Software	Generation phase	Analysis phase
Access 2007	41 s	16.86 ms
Excel 2007	36 s	15.66 ms
Word 2007	35 s	25.75 ms
PowerPoint 2007	46 s	26.74 ms
Outlook 2007	80 s	23.33 ms
Visio 2007	35 s	15.28 ms
Onenote 2007	29 s	11.68 ms

Table 11: Execution time for analyzing MS Office 2007.

Document with component Table 9 shows our discovered "document with component" attacks. As we mentioned above, third-party components can cause serious security vulnerabilities in software. According to the Table, Microsoft Word and PowerPoint 2007 suffer from security vulnerabilities caused by third-party components, which lead to remote code execution attacks. In particular, loading the `Google Desktop Office Addin` and the `HP printer driver` fails to resolve particular DLLs when the programs open documents, and the current directory at that point is the same directory as the opened documents. This security hole allows attackers to make the software load the DLLs from remote sites when the victim opens the document. We reported this issue to the Microsoft Security Response Center, and we are working with Microsoft in collaboration with Google and HP to develop security patches.

For Foxit Reader 3.0, the directory containing the opened PDF document can be considered the DLL-hijacking directory due to a resolution failure of `rpawinet.dll` in Windows Vista. The unsafe component loading is performed at the point of checking for updates. Note that updates are checked by the application automatically. Based on this flaw, attackers can perform remote code execution attacks by sending archives of a PDF document and a malicious `rpawinet.dll` to remote users.

Combination with the Carpet Bomb attack In Windows Vista Home Edition SP1, Internet Explorer 8 fails to resolve the DLL `rpawinet.dll` on its startup, and the current directory at that point is always the Desktop directory (Table 10). This unsafe behavior can lead to blended attacks combined with the Safari's Carpet bomb attack.

5.3 Performance

To evaluate the performance of our technique, we measure the execution time of each phase for analyzing MS Office products on Windows XP SP3 running on a Core2 Duo 2.40GHz processor with 4GB RAM. Table 11 shows the execution time for the generation and analysis phases of the analyzed applications. In the evaluation, we use default documents as inputs to the analyzed programs. Our results show that our technique is practical and can be effectively applied for analyzing real-world programs such as MS Office.

6. DISCUSSION

In this section, we discuss techniques to mitigate unsafe dynamic loadings, generality of our proposed technique, and unsafe component loading as a general security concern.

6.1 Mitigation Techniques

Use fullpath Because the filename specification resolves the target component by iterating through the directories, it may lead to resolution hijacking. This problem can be solved by specifying the target DLL based on its full path, because the fullpath specification determines its target file directly without iteratively searching a set of directories. In order to generate correct fullpath specifications, system calls that return full paths of the target directories can be used. For example, suppose a developer wants to load a DLL in the system directory at runtime on Microsoft Windows. In this case, `GetSystemDirectory` function can be used to determine the full path of the DLL. In particular, after obtaining the path of the system directory through the system call, the developer can concatenate the path with the filename of target DLL to obtain its full path. For instance, if a developer wants to load `WS2HELP.DLL` in the system directory, safe DLL resolution can be achieved by concatenating `WS2HELP.DLL` with the system directory path obtained by the `GetSystemDirectory` function (*i.e.*, `C:\Windows\System32`).

Resolve system call at runtime According to Section 5, chained loading of DLLs also causes resolution hijacking. This can be mitigated by resolving system calls at runtime as much as possible. In particular, if we resolve the address of the target system call exported by a DLL and invoke it at runtime, the DLL file is not considered a dependent DLL and is not loaded at load-time. For example, suppose we want to invoke the `send` function of `ws2_32.dll`, we can obtain the function's address by using the `LoadLibrary` and `GetProcAddress` functions exported by `kernel32.dll` at runtime, and invoke the target function based on this address.

Confirm file existence As we mentioned in Section 5, resolution failures can cause serious security vulnerabilities in software. Its main reason is that many programs make the false assumption that the target component exists in the system. Therefore, it is important to check existence of the target files before loading them.

Check current OS version As we discussed in Section 5, a set of system libraries depends on the version of the operating system. Because many Windows applications are developed to be executable under both Windows XP and Windows Vista, they should check version of the OS and load only the supported components.

Provide tools for checking third-party components Unsafe component loadings performed by third-party components can lead to serious security holes in the applications hosting them. Because of this security issue, although the applications resolve the components safely, they can be attacked by exploiting vulnerabilities in the third-party components. To mitigate this problem, it is necessary for application developers to provide developers of the third-party components with tools to check the safety of their components.

Check validity of loaded DLLs Because a program resolves a target DLL based on its name, it is difficult to determine whether or not the resolved DLL is the file intended by the program. To address this problem, application developers can use properties of the target file such as its hash value to determine validity of the loaded component.

Use SetDllDirectory function As we mentioned in Section 5, the current directory at the point of a resolution failure may cause remote code execution attacks. To mitigate this type of attacks, we can use the `SetDllDirectory` function which can add an arbitrary directory instead of the current directory. Especially, this function can remove the current directory from the directory search order. This approach can effectively block remote code execution attacks discussed in Section 2.3. In particular, Microsoft adopts this approach to fix the blended attack combined with the Safari's Carpet Bomb attack [28].

Install applications in the admin-writable directory According to Table 6, the application directories are the most vulnerable ones to resolution hijacking. Therefore, unsafe resolutions performed by non-admin users can be significantly reduced by installing applications in directories only writable by administrators (*e.g.*, the `Program Files` directory on Microsoft Windows).

6.2 Generality of Our Approach

Although we have focused on detecting and analyzing unsafe component loadings on Microsoft Windows, the underlying principle is general and can also be applied to other operating systems. For Unix-like operating systems (*e.g.*, MAC OS X, Solaris, and Linux), dynamic loading is performed by the `dlopen` system call. According to its man page [11], target component resolution on these operating systems is done in a similar manner as what is done on Microsoft Windows as we discussed in Section 2. In particular, the fullpath specification starting with '`/`' determines the target component directly, while the filename specification iterates through a list of predefined directories to find the file with the specified name. Furthermore, chained loading happens based on dependencies of the target component [22].

The first phase of our technique is platform-dependent, while the second is platform-independent. Thus, to adapt the technique to a different operating system, we need to specialize the generation of dynamic profiles w.r.t. its component loading mechanism. However, the offline profile analysis is platform-independent, and a generic implementation is possible to process the generated dynamic profiles and detect unsafe component loadings.

6.3 Unsafe Loading as A General Concern

Unsafe component loading is essentially a type of programming defects [17, 31]. Therefore, this problem often arises in operating systems that support dynamic loading. As a recent example, CVE-2009-0415 [9] is a vulnerability due to unsafe component loading. In particular, when Trickle [35], a user space bandwidth shaper for Unix-like systems, loads `trickle-overload.so`, it checks the current working directory before the intended resolution (*i.e.*, `/usr/lib/trickle/trickle-overload.so`). This flaw allows local attackers to execute arbitrary code by using malicious `trickle-overload.so` in the current working directory.

7. RELATED WORK

This section surveys closely related work and divides the related work into four categories: framework for safe component resolution, safety improvement of browser plugins, vulnerability analysis and detection, and non-control-data attacks.

Chari *et al.* [7] presents a mechanism, `safe-open`, to prevent unsafe component resolution in Unix by detecting modifications to path names by untrusted users on the system. In comparison, we propose a dynamic analysis to discover unsafe component loading vulnerabilities in the software itself.

Secure browsers [14, 15, 16, 36] have been introduced to mitigate risks caused by unsafe usage of third-party plug-ins. Gazelle [36] and OP [16] browsers adopt OS-level sandboxing techniques to reduce damages introduced by unsafe plugin usage. Grier *et al.* [15] propose security policies for secure plugin execution. Internet Explorer utilizes a kill-bit [20] to prevent malicious ActiveX components from being loaded. These techniques aim at providing software platforms with secure plugin usage, while our technique aims at detecting unsafe loadings of general software components.

Testing and analysis techniques for detecting software vulnerabilities have been well explored. Most of previous approaches have focused on detecting low-level, unexpected program behaviors such as memory corruption errors [5, 6, 10, 13, 21, 29, 30, 41] and integer overflows [4, 27, 37]. Although these approaches have shown

promising results in detecting such vulnerabilities, none has targeted the detection of unsafe component loadings; our work formulates the problem and introduces the first effective automated technique to detect such vulnerabilities.

Unsafe component loading can also be considered an example of non-control-data attacks because it does not alter the control data of the target program. Chen *et al.* [8] surveyed attack techniques that corrupt application data, which includes user identity data, configuration data, user input data, and decision-making data, and presented a detailed analysis and defense mechanism. Compared to those non-control-data attacks, unsafe dynamic loading is mainly due to defects in the component loading procedure, while they are originated from unsafe handling of application data. In addition, the attack vectors are different. In particular, unsafe component loading can be exploited by placing malicious files in the component-hijacking directories, while non-control-data attacks corrupt certain application data to exploit unsafe processing of the data.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a dynamic analysis technique to detect unsafe dynamic component loadings. Our technique works in two phases. It first generates profiles to record a sequence of component loading behaviors at runtime using dynamic binary instrumentation. It then analyzes the profiles to detect two types of unsafe component loadings: resolution failures and resolution hijackings. To evaluate our technique, we implemented a tool to detect unsafe DLL loadings on Microsoft Windows. Our evaluation shows that unsafe DLL loading is prevalent and can lead to serious threats. In particular, our tool detected more than 1,700 unsafe DLL loadings in popular software developed by major vendors. It also discovered potential remote code execution attacks exploiting the detected unsafe DLL loadings.

For future work, we are interested in exploring two research directions. First, we plan to analyze unsafe component loadings in Unix-like operating systems. As we mentioned in Section 6, unsafe component loading is a general security concern, and our approach is general and can also be applied to analyze applications on these systems. We plan to evaluate the prevalence and severity of unsafe component loading for these other important operating systems. Second, we plan to develop static binary analysis techniques to detect unsafe component loadings. Although our dynamic analysis is effective, it may suffer from the standard limitation of dynamic analysis, namely the code coverage problem. We plan to develop sound, practical static analysis techniques to complement the dynamic analysis we introduced here.

Acknowledgments

We thank Earl Barr, David Hamilton, David LeBlanc, and the anonymous reviewers for useful feedback on earlier versions of this paper. We also thank Charles Weidner at MSRC for collaborating with us to develop patches for the reported vulnerabilities.

References

[1] About the security content of Safari 3.1.2 for Windows. `http://support.apple.com/kb/HT2092`.

[2] About Windows Resource Protection. `http://msdn.microsoft.com/en-us/library/aa382503(VS.85).aspx`.

[3] About Windows Resource Protection. `http://www.dependencywalker.com/help/html/dependency_types.htm`.

[4] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *Proc. NDSS*, 2007.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proc. CCS*, 2006.

[7] S. Chari, S. Halevi, and W. Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *Proc. NDSS*, 2010.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proc. Usenix Security Symposium*, 2005.

[9] CVE-2009-0415. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0415`.

[10] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. ICSE*, 2006.

[11] dlopen man page. `http://linux.die.net/man/3/dlopen`.

[12] Dynamic-Link Library Search Order. `http://msdn.microsoft.com/en-us/library/ms682586(VS.85).aspx`.

[13] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, 2008.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the Wily Hacker. In *Proc. Usenix Security Symposium*, 1996.

[15] C. Grier, S. T. King, and D. Wallach. How I learned to stop worrying and love plugins. In *Proc. W2SP*, 2009.

[16] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proc. S&P*, 2008.

[17] T. Grtker, U. Holtmann, H. Keding, and M. Wloka. *The Developer's Guide to Debugging*. Springer, 2008.

[18] IDA Pro Disassembler. `http://www.hex-rays.com/idapro/`.

[19] IE's unsafe DLL loading. `http://www.milw0rm.com/exploits/2929`.

[20] Killbit. `http://support.microsoft.com/kb/240797`.

[21] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. Usenix Security Symposium*, 2001.

[22] ldd man page. `http://linux.die.net/man/1/ldd`.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005.

[24] Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[25] Microsoft Security Bulletin MS09-014. `http://www.microsoft.com/technet/security/Bulletin/MS09-014.mspx`.

[26] Microsoft Security Bulletin MS09-015. `http://www.microsoft.com/technet/security/Bulletin/MS09-015.mspx`.

[27] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proc. Usenix Security Symposium*, 2009.

[28] MS09-014: Addressing the Safari Carpet Bomb vulnerability. `http://blogs.technet.com/srd/archive/2009/04/14/ms09-014-addressing-the-safari-carpet-bomb-vulnerability.aspx`.

[29] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. NDSS*, 2004.

[30] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. ISSTA*, 2009.

[31] Secure Linux Programming. `https://foss.in/2006/cfp/slides/Secure_Linux_Programming_145.pdf`.

[32] Side-by-side Assemblies. `http://msdn.microsoft.com/en-us/library/aa376307(VS.85).aspx`.

[33] The End of DLL Hell. `http://msdn.microsoft.com/en-us/library/ms811694.aspx`.

[34] The Long-Term Impact of User Account Control. `http://technet.microsoft.com/en-us/magazine/2007.09.securitywatch.aspx`.

[35] Trickle. `http://monkey.org/~marius/pages/?page=trickle`.

[36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proc. Usenix Security Symposium*, 2009.

[37] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. NDSS*, 2009.

[38] What Goes On Inside Windows 2000: Solving the Mysteries of the Loader. `http://msdn.microsoft.com/en-us/magazine/cc301727.aspx`.

[39] O. Whitehouse. GS and ASLR in Windows Vista. In *Black Hat DC*, 2007.

[40] Windows Management Instrumentation. `http://msdn.microsoft.com/en-us/library/aa394582(VS.85).aspx`.

[41] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proc. ISSTA*, 2008.