Using Labeling to Prevent Cross-Service Attacks Against Smart Phones

Collin Mulliner[1], Giovanni Vigna[1], David Dagon[2], and Wenke Lee[2]

[1] University of California, Santa Barbara, USA
{mulliner, vigna}@cs.ucsb.edu
[2] Georgia Institute of Technology, Atlanta, USA
{dagon, wenke}@cc.gatech.edu

Abstract. Wireless devices that integrate the functionality of PDAs and cell phones are becoming commonplace, making different types of network services available to mobile applications. However, the integration of different services allows an attacker to cross service boundaries. For example, an attack carried out through the wireless network interface may eventually provide access to the phone functionality. This type of attacks can cause considerable damage because some of the services (e.g., the GSM-based services) charge the user based on the traffic or time of use. In this paper, we demonstrate the feasibility of these attacks by developing a proof-of-concept exploit that crosses service boundaries. To address these security issues, we developed a solution based on resource labeling. We modified the kernel of an integrated wireless device so that processes and files are marked in a way that allows one to regulate the access to different system resources. Labels are set when certain network services are accessed. The labeling is then transferred between processes and system resources as a result of either access or execution. We also defined a language for creating labeling rules, and demonstrated how the system can be used to prevent attacks that attempt to cross service boundaries. Experimental evaluation shows that the implementation introduces little overhead. Our security solution is orthogonal to other protection schemes and provides a critical defense for the growing problem of cell phone viruses and worms.

1 Introduction

Mobile devices such as Personal Digital Assistants (PDAs) and cell phones are converging. The new devices created through this convergence integrate different wireless technologies such as IEEE 802.11, Bluetooth, and GSM/GPRS. Unfortunately, the integration of different network services is often performed by simply including the necessary hardware and software components in a single device, without considering the different characteristics of each technology and the services bound to them. As a result, highly-integrated devices may be vulnerable to a novel class of attacks that leverage the interaction between different services.

R. Büschkes and P. Laskov (Eds.): DIMVA 2006, LNCS 4064, pp. 91–108, 2006.

A particularly notable example is the interaction between free services and subscription-based services. Cell phones are bound to carriers through a service agreement where the user is billed by the time spent using the service and/or by the amount of data transferred. PDAs, on the other hand, usually support (free) access to both wireless and wired IP-based local area networks (LANs). Although cell phone service providers implement firewalls and other forms of protection to safeguard the security of users' devices, little protection is provided when accessing wireless or wired LANs. Therefore, an integrated device may be compromised by exploiting the local area network connectivity and leveraged to access subscription-based services, causing monetary loss to the user.

This situation is worsened by the improved storage and computational power provided by integrated devices. The availability of relatively high-performance PDA platforms support the execution of third-party, network-accessible services (e.g., personal databases and network file servers), which increase the security exposure of the device. In addition, these network-based applications are often developed without much concern about security and without considering the possible interaction between different network services.

To demonstrate the feasibility of sophisticated attacks against devices that integrate cell phone and PDA functionality, we developed a proof-of-concept attack, where a buffer overflow vulnerability in a network-accessible service is exploited through the 802.11b wireless interface. The malicious payload executed as a result of the attack is then able to access the cell phone functionality and place (possibly expensive) phone calls on behalf of the attacker. Even though buffer overflow attacks are not a new concept, to the best of our knowledge, this is the first detailed description of what a cross-service attack entails, including some non-trivial aspects of the exploitation.

The current security mechanisms deployed in integrated mobile devices do not provide any protection against this type of attacks. To address the security issues associated with integrated devices that can access multiple network services, we devised a novel mechanism to compartmentalize the access to system resources. The overall goal of our mechanism is to prevent processes that interacted with a particular network service (e.g., the wireless IP-based network) from crossing the service boundaries and access the resources associated with different services (e.g., the GSM-based services).

Our mechanism monitors the system calls executed by running processes and labels executing code based on its access to the network interfaces (e.g., wireless, GSM, Bluetooth). The labeling is then transferred between processes and system resources as a consequence of either access or execution. When sensitive operations are performed, the labels of the involved resources (processes and/or files) are compared to a set of rules. The rules allow one to specify fine-grained access control to services and data. For example, it is possible to restrict the access of an address book application to the phone dialing API, and, in addition, prohibit access to unrelated APIs (e.g., the socket API). The labeling of processes and resources, as well as the enforcement of the policies, are performed by a kernel-level reference monitor.

To make our mechanism general and easily configurable, we defined a policy language that allows one to express what actions are allowed by specific classes of programs with respect to specific classes of resources.

To demonstrate the usability of our mechanism, we implemented a prototype of the labeling system and the associated reference monitor on the Familiar Linux [11] platform. We also experimentally evaluated the overhead introduced by the mechanism.

The rest of this paper is structured as follows. Section 2 describes our proof-of-concept attack against devices that integrate PDA and cell phone functionality. Section 3 illustrates the design of our labeling mechanism. Then, Section 4 describes the details of our prototype implementation. Section 5 presents the experimental evaluation of our security mechanism in terms of both its effectiveness in preventing cross-service attacks and the overhead introduced. Then, Section 6 discusses related work and Section 7 briefly concludes.

2 A Proof-of-Concept Cross-Service Attack

We implemented a proof-of-concept attack that shows how it is possible to first break into a cell phone/PDA integrated device by means of its wireless LAN interface and then access the device's phone interface to dial a number. The attack was performed against a Pocket PC-based integrated device [22]. The proof-of-concept attack has been developed against two targets. The first is an application we developed to easily demonstrate the attack; the second is a 0-day attack against a real-world application. Note that this attack has not been made public yet.

2.1 An Attack Scenario

The proof-of-concept attack is an "over-charging" attack against the subscription-based service of a user, where the victim's cell phone is leveraged to place expensive phone calls (e.g., to a pay-per-minute 900 number). Other attacks are possible, but the fact that over-charging attacks may generate a revenue for the attacker (and a loss for the victim) suggests that they have the potential of becoming widespread soon.

To illustrate an instance of the attack, one can imagine a traveling salesman who walks into a coffee shop seeking wireless Internet access in order to check his corporate email and online calendar. The salesman starts his integrated cell phone/PDA and associates the wireless LAN interface on his device with the coffee shop's wireless access point.

The attacker is monitoring the coffee shop's wireless network and sees the new device associating with the access point. Therefore, he immediately scans the new device and discovers a well-known vulnerable service. Using an exploit previously published on a security mailing list for the identified service, the attacker gains access to the phone. The exploit payload contains code that dials a 900 number owned by the attacker, charging hundreds of dollars to the victim's account.

2.2 The i-mate PDA2k Phone

To demonstrate the above scenario, we use the i-mate PDA2k [17], an OEM version of the HTC Blue Angel [16], a so-called "smart phone" running the Windows Mobile 2003 Second Edition operating system. The device is based on an Intel XScale PXA263 processor, which is an ARM CPU. The device is equipped with a wireless LAN (802.11b) interface, a Bluetooth [4] interface, and multi-band GSM [14] and GPRS [13] services. We chose this device for our proof-of-concept attack because it represents the type of device that will become common in a few years. A picture of the device appears in Figure 1.

Fig. 1. The i-mate PDA2k

2.3 A Vulnerable Service

Buffer overflow vulnerabilities account for the vast majority of security exposures across all platforms. Therefore, we chose this type of attack for our example.

We started off with our own vulnerable application, a simple echo server (similar to the `echo` service on UN*X systems). The application accepts incoming connections and then echoes back the received data. The server fails to check the length of the received data when copying strings, and, therefore a buffer on the stack can be overflown with data that eventually hijacks the server's control flow.

To determine the likelihood of finding similar vulnerabilities against WindowsCE applications, we analyzed a number of applications, both in binary and source form. In particular, we focused on applications that listen for incoming

connections. For example, some Session Initiation Protocol (SIP) tools [24] listen for incoming Internet phone calls on port 1720 [30]. Likewise, multiple HTTP [23] and FTP servers [33, 8] are available for WindowsCE. Several of these applications obviously don't perform correct length checks on external input and crashed when stimulated with specially-crafted input data.

We chose `ftpsvr` [8], an open-source FTP server, as our target. We found that the server contains a buffer overflow vulnerability that can be exploited to achieve a cross-service attack. We provide more details about the vulnerability and the exploit in the next paragraph.

2.4 Exploiting the Vulnerability

The vulnerability we used for the attack is a simple `strcpy` attack in the function `void Session::SendToClient(int mode, LPCSTR msg)` in `ftpmain.cpp`. The function is called to respond to client commands, which, in some cases, echoes back data provided by the client. The attack utilizes the `USER` command and the error handler for unknown commands. Both operations utilize `SendToClient`, passing unchecked client input to it. The `strcpy` invocation inside `SendToClient` writes to a fixed-size buffer of 256 bytes, which allows one to overwrite the return address of the function's stack frame. Because of random memory corruption of old stack frames on function exit, we had to first upload the shellcode into a safe place. For this we utilized the `unknown command` error handler. The handler stores the string that doesn't match any command in the global variable `m_szSjis` just before sending an error to the client. Modification of the program counter is done by utilizing the `USER` command, which overwrites the return address with the address of `m_szSjis`.[1]

Using just the address of the shellcode as return address is not enough on WindowsCE. This is because the WindowsCE memory architecture [21] has only one virtual address space for the kernel, dynamic libraries and processes, with a maximum of 32 processes executing concurrently. Therefore, each process is placed in one of the pre-determined "slots" (pseudo-virtual address spaces). The slot number is determined by the most significant byte of an address. A particular case is represented by the current active process, which is mapped to slot 0 in addition to its actual slot. Note that in this case the most significant-byte is 0, and since the exploit needs to be zero-free in order to be processed by string functions, the actual slot must be found in order to successfully exploit the vulnerability.

Finding the right slot is not infeasible. First, the total number of slots is small (32); second, slots are assigned in order (bottom up); and third, system processes use fixed slots which further cuts down the search space. In addition, if a vulnerability in a system process is found, no search is required to exploit it, because the process uses a fixed slot.

Note that, using a wrong address will usually just lockup the target device, forcing the user to reset/reboot. After restart, the guessing becomes much easier

[1] For a general overview of how buffer overflows work, see [18].

since the target application will likely be placed in one of the lower memory slots.

Writing a malicious payload (i.e., the "shellcode") for WindowsCE is straightforward. The only complication comes from the requirement that only library calls can be used instead of system calls. Thus, one must additionally find the address of where in memory the desired library calls are mapped. This mapping information is device- and version-specific and can be gathered off-line. As a result, the attacker only needs to discover the device type to determine the correct address. This problem can be partially solved using the `WindowsCE API Address Search Technology` [27], which does the function address lookup on-the-fly, and, therefore, can produce portable shellcode. However, this technique introduces a substantial amount of overhead (in terms of shellcode size).

In most cases, using library calls in WindowsCE shellcode is straightforward, once the address of the target call is known. A call is done in four steps: first, the function address needs to be loaded into a register; second, the function parameters also need to be loaded into registers (for more than four parameters the stack is used to pass the additional parameters); third, the return address has to be saved to the `link register` (LR); in the fourth step, the call is executed by direct modification of the `program counter`, setting it to the address of the function.

Additional care needs to be taken to remove any zeros from the shellcode. This is a general problem when dealing with string functions. In addition, both the ARM architecture and WindowsCE add additional sources for zeros. ARM instructions have fixed length (4 bytes), and, therefore, some instructions will contain zero bytes (e.g., every time register r0 is used). As another example, WindowsCE uses mostly Unicode strings, which will add multiple zero bytes for each string. To remedy this problem we used a simple `XOR` encryption to remove zeros. Our shellcode contains a small bootstrap routine which decrypts the main payload, as it is often done with polymorphic malware.

Once the payload of the attack is executed, the code places a phone call. This is done in two steps. In the first step, the phone library is loaded (mapped) into the application's address space. This is done by calling `LoadLibraryW(TEXT ("cellcore"))`. In the second step, the phone call is executed by calling `tapiRequestMakeCall`, which dials the given number. The number is a Unicode string passed as the first parameter to `tapiRequestMakeCall`.

In summary, we were able to craft an exploit for the WindowsCE platform that overflows a buffer in a network-based application, and then forces the victim's device to place a phone call. Recent postings [6, 1, 2] to security lists like [29] underline our assumptions that exploits for WindowsCE will soon be publicly available, and, therefore, could be used as a vector for this type of attack.

3 Preventing Cross-Service Attacks Through Labeling

The exploit described in the previous section demonstrates how an attack can cross service boundaries and abuse the resources of an integrated cell phone/PDA

device. Traditional solutions, such as stack protection mechanisms [5], require compiler support and are not yet widely available for WindowsCE devices. Even though version 5.0 of the Microsoft WindowsCE build environment has an option to protect against stack-smashing attacks (i.e., the `/GS` option [20]), this feature is not enabled by default. Also, cross-service attacks can be carried out without performing buffer overflows (e.g., by exploiting application-logic errors), and, therefore, a solution directly targeted to prevent these attacks is needed.

To counter cross-service attacks, we developed a security mechanism based on process and system resource labeling. The mechanism defines three types of objects, namely *processes* $p_1, p_2, ..., p_n \in P$, *resources* $r_1, r_2, \ldots, r_m \in R$, and *interfaces* $i_1, i_2, \ldots, i_k \in I$. Processes and resources have an associated set of labels $l_1, l_2, \ldots, l_j \in L$. Each label represents the fact that, either directly or indirectly, the process or resource was in contact with a specific network interface. We define $L(i)$ the label associated with interface i. In addition, we represent with $LS(p)$ and $LS(r)$ the set of labels associated with a process p and a resource r, respectively.

Our security mechanism includes a monitoring component that intercepts the security-relevant system calls performed by processes. These are the system calls that access interfaces, access/execute resources, create resources, and create new processes. When a security-relevant system call is intercepted, the labels of the executing process are examined with respect to a global policy file that specifies which types of actions are permitted, given the labels associated with a process. The result of the analysis may be that the access is denied, that the access is granted, or that the access is granted and, in addition, the labels of the resource/process involved in the operation are modified. In the following, we present in more detail the operations performed by the labeling mechanisms in relation to the execution of certain types of system calls.

Interface access. When a process accesses an interface, the process' labels are examined to determine if access should be granted. If this is the case, the process gets marked with a label representing the specific interface being accessed, that is, $LS(p) = LS(p) \cup L(i)$, where p is the process accessing interface i. For example, if a process accessed the wireless LAN interface by performing a socket-related system call, then the process is marked with a label that specifies the wireless LAN interface.

Resource access. When a process requests access to a resource (for example, when trying to open a file) the labels associated with both the process and the resource are examined with respect to the existing policy. If access is granted, then the label set of the process is updated with the label of the resource, that is, $LS(p) = LS(p) \cup LS(r)$, where p and r are the process and the resource involved, respectively.

Resource and process creation. When a process p creates a new resource or modifies an existing one, say r, the resource inherits the label set of the process, that is $LS(r) = LS(p)$. In a similar way, when a process p creates a new process p' the labels are copied to the newly created process, that is, $LS(p') = LS(p)$.

The labeling behavior described above allows the security mechanism to keep track of which interfaces were involved and of which processes and resources were affected by security-relevant actions. For example, if a process bound to a certain interface was compromised, the files (or the processes) created by the compromised process will be marked with the label associated with the interface. When the compromised process (or a process that is either created by the compromised process or that accesses or executes a resource created by the compromised process) attempts to access other interfaces, it is possible to identify and block the attempt to cross a service boundary.

3.1 Policy Specification

The security mechanism uses a policy file to determine whether to grant or deny a process access to a resource or interface. In addition, the policy file can be used to modify the default labeling behavior described above.

Access control is performed by specifying which label or labels a process is not allowed to have when accessing a specific resource or interface. By default, access is granted to all interfaces and resources. Of course, this default policy is not very secure, but we anticipate that service providers will create comprehensive rules for their users, or that power users will adopt more restrictive rules, as needed.

The policy file consists of a set of rules, where a rule is composed of the target interface or resource, the action to be performed by the reference monitor when access is requested, and the labels that trigger the action. The access control language is defined as follows:

$policy \Rightarrow rule*$

$rule \Rightarrow$ `access` $(interface|resource)\ action\ label*$

$action \Rightarrow$ `deny`|`ask`

The `deny` action simply denies access, while the `ask` action prompts the user for confirmation through an interactive dialog box. For example a rule like:

```
access i1 deny i2 i3
```

would deny access to interface `i1` if the process was previously labeled with the labels associated with interfaces `i2` or `i3`.

As stated before, the policy file can also be used to modify the default labeling behavior. By default, every process becomes labeled when it accesses an interface (or another labeled resource) or when it is created by a marked process. The policy language can be used to define which applications are excluded from this behavior. We define three actions that modify marking in a certain way. The `notlabel` action denotes that the process executing the specified application is not labeled when touching an interface. The `notinherit` action denotes that the process does not inherit any labels when accessing objects. The `notpass` action

denotes that the process is not passing labels to resources and processes. These extensions to the policy language are defined as follows:

$rule \Rightarrow$ `exception` $path\ action*$

$path \Rightarrow$ / $(dirname/) * \ filename$

$action \Rightarrow$ `notlabel`|`notinherit`|`notpass`

The *path* variable specifies the file containing the application whose behavior has to be modified.

Consider, as an example, a rule for a trustworthy synchronization application that is used to transfer and install files to a device using the USB cable interface. The synchronization application needs access to the USB interface to operate correctly, and, at the same time, it is not desirable that all the files created by the application are labeled with the interface used for synchronization. Therefore, a set of `exception` rules for the synchronization application can be used to specify that the process is not marked with any label and does not inherit or pass labels to and from resources. In this case, the user can trust the synchronization application because it can operate only using the USB interface which requires physical access to the device. This is a somewhat over-simplifying example. Some synchronization operations may be performed through other interfaces such as Bluetooth or the Internet. In such cases, the policy should be modified accordingly. (In addition, a very security conscious user may even turn off Internet synchronization, and use Bluetooth judiciously.)

As another example, consider a rule for a Web browser which specifies that the process does not inherit labels from files. This is necessary, since the browser must access previously downloaded files (e.g., the browser cache). This prevents the browser from becoming labeled and possibly unable to access the network.

The `notpass` action can be used to specify which applications can create non-marked files. This mechanism can be used to implicitly remove labels from a file by making a copy of it using an application which has the `notpass` action set. An example is the FileExplorer application. A sample marking policy for PocketPC could look like the one showed in Figure 2, while a sample marking policy for a Familiar Linux installation may be similar to the one shown in Figure 3.

```
# Internet Explorer
exception /Windows/iexplore.exe notinherit

# ActiveSync
exception /Windows/repllog.exe notlabel notinherit notpass

# FileExplorer
exception /Windows/fexplorer.exe notpass
```

Fig. 2. Sample policy file for PocketPC

```
# Konqueror (web browser)
exception /opt/bin/konqueror notinherit

# Ipkg (package management tool)
exception /usr/bin/ipkg-cl notlabel notinherit notpass

# multi-purpose binary
exception /opt/QtPalmtop/bin/quicklauncher notpass notinherit
```

Fig. 3. Sample policy file for Familiar Linux

4 Implementation

Even though our proof-of-concept attack was against the WindowsCE OS, we implemented a prototype of our labeling system for the Familiar Linux distribution, because we needed to be able to modify the kernel of the operating system. We used the Familiar release `0.8.2` as our base system, and we modified the kernel and added a few utilities. The kernel version used was `2.4.19-rmk6-pxa1-hh37`. Like many other host-based monitoring approaches, our monitor runs in the operating system kernel, and it is safe from tampering unless the root account is compromised.

Our prototype monitors access to files and communication interfaces, such as the wireless LAN interface or the phone interface. Monitoring and enforcing the object marking is implemented by intercepting the system calls used to access the objects of interest and carrying out the actions specified by the policy rules. Program execution is handled through monitoring of the `execve(2)` system call. Network related access is monitored through the `socket(2)` family of system calls. File system monitoring, including device files (e.g., serial line device), is done by intercepting the `open(2)` system call. We also added to the kernel additional system calls for loading labeling and exception polices into kernel space.

Processes are marked with a label by the monitor upon accessing either a monitored interface or a file in the filesystem. The labels are implemented as bits in a bit-field, shown in Figure 4, which is stored in the process descriptor structure of the operating system kernel. Each label in the bit-field represents a specific communication interface. When a process attempts to access a system resource, the relevant labels are checked against a kernel-resident data structure containing the policy.

Files created or touched by a marked process inherit the process' labels (as explained in Section 3, this "tainting" process also works in the other direction). File marking is implemented by adding the same bit-field used for process labels to the file structure in the filesystem. This is done by maintaining file-specific data structures in the operating system kernel.

Labels are used to specify the interfaces in a device that provide some kind of communication with the outside world. In our implementation, labels are

divided into three subsets. This classification provides a more general way to define access policies.

Wired. This set of labels contains all interfaces which need some kind of physical connection in order to communicate. Example devices include: the serial interfaces, USB interfaces, and Ethernet interfaces.

WirelessNonfree. This set of labels contains all wireless interfaces bound to a subscription-based service. Examples are: GPRS, GSM_voice, and GSM_data.

WirelessFree. This set of labels contains interfaces that are not bound to a subscription-based service. Examples include Infrared, Bluetooth_voice, Bluetooth_data, and Wi-Fi.

	wired
0	serial
1	USB
2	Ethernet
3	
4	
	wireless non-free
5	GSM voice
6	GSM data
7	GPRS
8	
9	
	wireless free
10	Wi-Fi
11	Bluetooth voice
12	Bluetooth data
13	Infrared

Fig. 4. Label bit-field

Given the set of labels defined in Figure 4, the policy language of our prototype can be further defined as follows:

$interface \Rightarrow wireless\_nonfree|wireless\_free|wired$

$wireless\_nonfree \Rightarrow$ `gsm_voice`|`gsm_data`|`gprs`

$wireless\_free \Rightarrow$ `infrared`|`wifi`|`bluetooth_voice`|`bluetooth_data`

$wired \Rightarrow$ `serial`|`usb`|`ethernet`

$label \Rightarrow wired|wireless\_nonfree|wireless\_free$

The rule language is expressive and powerful enough to stop many types of cross-service attacks. For example, a rule preventing the proof-of-concept attack described in Section 1 would look like:

```
access wireless_nonfree deny wireless_free
```

This rule denies access to all non-free wireless interfaces to processes which have touched any of the free wireless interfaces. It would still permit processes compromised through free interfaces to access other free interfaces. However, this simple one-line rule would permit flexible use of a device, with the assurance that an attack would not result in additional service billing or cost charges. If a more restrictive rule is required, the policy language permits users and/or service providers to further lock down the system.

Note that although it cannot stop all types of attacks, the labeling system addresses operations at a semantic and functional level. This way, new attacks can be remedied quickly by modifying the set of policy rules. Other orthogonal solutions, such as stack protection or traditional IDSs, can also be used, but, as noted above, these solutions are either expensive for handhelds, or are not yet widely available. Therefore, our labeling solution provides an effective defense for integrated cell phone/PDA devices.

5 Evaluation

The device used to evaluate our system is an HP iPAQ h5500 [15] which is ARM-based, like the i-mate device, and runs Familiar Linux.

To test our solution, we first implemented the same proof-of-concept vulnerable `echo` server for the Linux OS. We then developed an exploit in a way similar to the one described in Section 2.4.

The access control policy used in the evaluation is the same as discussed in Section 4. The policy simply denies access to all non free wireless interfaces for processes that touched any free wireless interface.

5.1 Preventing the Attack

We will discuss the execution steps of the exploit to demonstrate how the labeling system prevents the attack. The `echo` server process is labeled upon creation of a socket (that is, when the process invokes `socket(AF_INET, ...)`). Since one cannot easily determine which interface will be used for IP networking, as a result of the socket operation both the label bits associated with *Wi-Fi* and *Ethernet* are set, covering both the free wireless and the wired class.

When the exploit code tries to access the port associated with the GSM interface using an `open(2)` system call, the reference monitor is invoked. The reference monitor then compares the process' bit-field with the rules specified in the policy file. The monitor denies access to the device, and the call to `open(2)` fails, returning `EACCESS`. Note that the buffer overflow may still take place, and the vulnerable application may likely crash. However, the over-charging attack cannot be performed.

As noted above, stack integrity protections and other orthogonal solutions can help prevent the buffer overflow in the first place. However, there are other types of vulnerabilities, e.g., application logic errors, to which these techniques are not applicable. Our policy labeling solution is general, simple, and efficient.

It gives assurances that attacks have limited impact, and will not result in the crossing of network services, which might cause billing charges.

5.2 Preventing Exploitation of Legal Privileges

Exploiting legal privileges of applications is a common method for circumventing access control mechanisms. In our system, this exploitation is prevented through the label inheritance on process creation. A newly created process will always inherit all labels from its creator, and, therefore, an attacker cannot use a new process to get rid of the labels and abuse his/her privileges.

If an application with legal access to a critical interface has the `notinherit` exception set, the protection is circumvented. Therefore, caution has to be taken when creating exception rules.

5.3 Accessing Multiple Interfaces Legally

The special case where an application needs to access multiple interfaces of different classes (specified in Section 4) could be problematic for our system.

An example for this kind of situation is a phone application which needs to access the GSM interface and Bluetooth in order to use a wireless headset for hands-free speaking. Another example would be roaming in next-generation telephony networks, where a phone application may need to access both the wireless LAN and the GSM interfaces.

These kind of situations can be handled through the use of a `notlabel` exception rule for specific applications. The rule will prohibit the labeling of the applications' processes when accessing any of the interfaces, and, therefore, these applications will be able to access all classes of interfaces. Note that processes will still inherit labels from accessed resources and from the parent process.

In summary, our system cannot detect attacks against applications that cross service boundaries by design. This is because the applications normal behavior matches the semantics of a service-crossing attack. We acknowledge this as an obvious shortcoming of our system. However, we believe that our mechanism still provides effective protection in most cases.

5.4 Overhead

One of our design goals was the creation of an efficient security solution, to encourage wide adoption. To evaluate the efficiency of our mechanism we measured the overhead introduced by the labeling system in two areas: the actual labeling and the access control enforcement.

Labeling Overhead. Executing a new application involves three steps: first, checking the *marking policy* for any special rules that might apply to the application being executed by the process; second, updating the process' bit-field (in particular clearing all labels if the marking policy specifies `notinherit`); third, checking the bit-field of the application's binary file itself (which is skipped if the marking policy specifies `notinherit`).

Further overhead is added through calls to `open(2)`. In this case, labels are inherited by the process and/or are passed to the file, depending on the process' exception rules and the open mode of the file. Calls to the `socket(2)` system call only add very little overhead, since only the exception rules need to be checked before the process is labeled.

For example, when the `wget` application is executed, the monitor is triggered by the `execve(2)` system call, which then performs the initial steps. Later, the monitor is triggered again, because of network and filesystem access (i.e., calls to `socket(2)` and `open(2)`, respectively).

Enforcement Overhead. The labeling system has a second potential impact on performance during enforcement. When enforcing a rule, the monitor has to compare the label bit-field of the process and the involved resource with the labels specified for each rule in the global policy. The monitor stops the analysis as soon as a matching `deny` rule is found.

For example, when the `ftp` application calls `socket(2)`, the monitor is triggered and searches the global policy for a rule matching the process' labels to decide if network access is to be granted, and, therefore, the socket can be created.

To measure the overhead introduced by our labeling system we chose three classes of tests: first, file access only; second, light network usage; third, heavy network usage. We used the `time` command to measure the time spent in the kernel during system calls. All tests were conducted using both the original kernel that came with Familiar and our own modified kernel.

To measure the overhead added to applications with only file access we ran `grep` on a directory containing 61 files and directories. In this test, 435 system calls were made with 1 call to `execve(2)` and 63 to `open(2)`. Intercepting the `open(2)` system call introduced some overhead. In the case of the `grep` test the overhead was 19%.

Measuring the overhead for applications with light network usage was done using `wget` to retrieve a file from a web server. Also, files are created (written to), and, therefore, labels are inherited from the `wget` process. In this test, 118 system calls were made with 1 call to `execve(2)`, 20 calls to `open(2)` and 1 call to `socket(2)`. Since `wget` only performs a few system calls which are intercepted, the introduced overhead of 26% mostly originates from the checks done within `execve(2)`.

For measuring the overhead for a heavy-weight network application we used `ncftpget` to download an entire directory (20 files) from a ftp server. In this test, 2220 system calls were made with 1 call to `execve(2)`, 54 calls to `open(2)` and 28 calls to `socket(2)`. Note that this test shows an overhead of only 10%. This is due to the fact that the startup penalty, introduced by the interception of `execve(2)`, is distributed over a longer execution time.

The results for all tests are shown in Figure 5. Note that the implementation of this prototype system is far from optimal. In particular, the implementation of the `open(2)` monitor has some performance issues. Overall, we are confident that

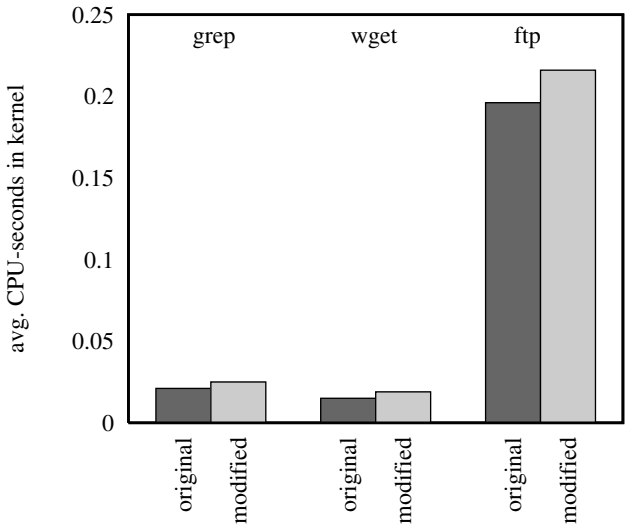

Fig. 5. Overhead evaluation

the overhead introduced by our system is small enough to provide a light-weight solution against cross-service attacks.

6 Related Work

Labeling processes to perform network access control is not novel, and similar techniques are often found in information assurance systems. For a comprehensive overview of information-flow security, see [26]. Our work is different from classic solutions of information-flow security, because our system tracks executable code instead of data. This prevents cross-interface exploitation and provides data protection.

Our work also fits into the larger field of access control [28]. Our work is similar to [9], where the authors created *Deeds*, a history-based access control system for mobile code. *Deeds* works with browser-based mobile code, and tracks dynamic resource requests to further differentiate between trusted and untrusted code. Our system is different in that the access policies are static, and not limited to just browser-based programs. Our use of static rules is appropriate to the handheld environment, where there are fewer applications than on a desktop.

Other labeling systems have been proposed. But since they were designed for desktop or server systems, they are too feature-heavy and introduce substantial administrative and performance overhead. Typical examples include hardened operating systems such as [19] and [34].

Our system shares similarities with LOMAC [12] which implements a form of low watermark integrity [3]. The difference is that our system distinguishes between different types of network interfaces. In the current implementation,

we mainly focus on the cost factor of different interfaces. Other factors like trustworthiness could be used instead.

Other security systems specifically target mobile devices, such as Umbrella [25]. Umbrella is a protection system based on signed binaries and mandatory access control mechanisms. It also heavily relies on the developers to write secure code. By contrast, our system presumes that some vulnerabilities will exist, and seeks to contain the impact of the attack on existing resources.

In the past year, viruses and worms targeting cell phones have started to appear in the wild [31]. Most of these viruses are either harmless proof-of-concepts, or need user interaction in order to infect a target. Some recent cell phone viruses, however, are malicious and destroy or degrade system resources [32, 10]. We believe that viruses targeting cell phones will soon become a major problem for consumers [7]. The interface labeling system we describe can help preventing not only directed break-in attacks but also the spread of worms and viruses targeting cell phones.

7 Conclusions

Much research needs to be carried out in the field of mobile device security. Our paper is the first in this area to demonstrate a cross-service vulnerability, and to propose a solution. Many of the problems found on desktop systems are starting to appear on handhelds. However, architectural differences between handhelds and desktops (e.g., less memory) present challenges for security designers.

We have designed and implemented an efficient labeling system to help mitigate or prevent cross-service attacks. Our prototype labeling system can be extended to effectively protect mobile devices against various threats. Future work will concentrate on extending the policy language to allow a user to describe more complex labeling policies and on making the implementation of the reference monitor more efficient.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

References

1. Airscanner Corp. Advisory 05081102 vxFtpSrv 0.9.7 Remote Code Execution Vulnerability. `http://www.airscanner.com/security/05081102_vxftpsrv.htm`, 2005.
2. Airscanner Corp. Advisory 05081203 vxTftpSrv 1.7.0 Remote Code Execution Vulnerability. `http://www.airscanner.com/security/05081203_vxtftpsrv.htm`, 2005.

3. K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.
4. Bluetooth SIG. Bluetooth. http://www.bluetooth.org, 2006.
5. C. Cowan, C. Pu, and D. Maier. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.
6. D. Elser. PicoWebServer Remote Unicode Stack Overflow Vulnerability. http://seclists.org/lists/bugtraq/2005/May/0333.html, May 2005.
7. D. Dagon, T. Martin, and T. Starner. Mobile Phones as Computing Devices: The Viruses are Coming! *IEEE Pervasive Computing*, October/December 2004.
8. E. Ito. FtpSvr - Ftp Server. http://www.oohito.com/wince/arm_j.htm, 1999.
9. G. Edjlali, A. Acharya, and V. Chaudhary. History-based Access Control for Mobile Code. In *ACM Conference on Computer and Communication Security*, 1998.
10. F-Secure Corporation. F-Secure Virus Descriptions : Skulls. http://www.f-secure.com/v-descs/skulls.shtml, 2004.
11. Familiar Linux - A Linux Distribution For Handheld Devices. http://familiar.handhelds.org/, 2006.
12. T. Fraser. LOMAC: MAC you can live with. In *Proc. of the 2001 Usenix Annual Technical Conference*, Jun 2001.
13. GSMA. GPRS - General Packet Radio Service. http://www.gsmworld.com, 2006.
14. GSMA. GSM - Global System for Mobile Communications. http://gsmworld.com, 2006.
15. Hewlett-Packard. HP iPAQ h5500. http://welcome.hp.com/country/us/en/prodserv/handheld.html, 2006.
16. HTC. HTC Blue Angel. http://www.htc.com.tw, 2006.
17. i-mate. i-mate PDA2k. http://imate.com/t-DETAILSP_DA2K.aspx, 2006.
18. J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2003.
19. P. Loscocco and S. Smalley. Integrating Exible Support For Security Policies Into The Linux Operating System. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, 2001.
20. Microsoft. Platform Builder for WindowsCE 5.0, Compiler Option Reference. http://msdn.microsoft.com/library/default.asp?url=/library/enus/wcepbguide5/ html/wce50congs-enablesecuritychecks.asp, 2005.
21. Microsoft. Microsoft WindowsCE .NET 4.2 Platform, Memory Architecture. http://msdn.microsoft.com/library/default.asp?url=/library/enus/wcema-in4/html/_wcesdk_windows_ce_memory_architecture.asp, 2006.
22. Microsoft. Windows Mobile. http://www.microsoft.com/windowsmobile/pocketpc/, 2006.
23. Newmad Technologies AB. PicoWebServer. http://www.newmad.se/rnd-freeswpico.htm, 2005.
24. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC3261, 2002.
25. M. T. S. N. Christensen, K. Sorensen. Umbrella - We can't prevent the rain ... -But we don't get wet! Master's thesis, Aalborg University, January 2005.
26. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
27. San. Hacking Windows CE. *Phrack*, 0x0b(0x3f), August 2005.
28. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

29. SecurityFocus. BugTraq. http://www.securityfocus.com/archive, 2006.
30. SJ Labs, Inc. Voice Over IP Software. http://www.sjlabs.com, 2005.
31. Symantec Security Response. SymbOS.Cabir. http://securityresponse.symantec.com/avcenter/venc/data/epoc.cabir.html, 2004.
32. Symbian, Inc. Information about Mosquitos Trojan. http://www.symbian.com/press-office/2004/pr040810.html, 2004.
33. Vieka Technology Inc. PE FTP Server. http://www.vieka.com/peftpd.htm, 2005.
34. R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *USENIX Annual Technical Conference, FREENIX Track*, pages 15–28, 2001.