Java 2 Micro Edition security vulnerabilities

Discovered, researched and presented by

ADAM GOWDIAK

HITB Security Conference, 4-7th October 2004, Kuala Lumpur, Malaysia

INTRODUCTION

ABOUT PSNC

- The leading research, network and supercomputing facility in Poland
- Operator of the Metropolitan Area Network and Polish National Academic Network (PIONIER)
- Leader of many research projects funded by the Polish State Committee for Scientific Research and European Community
- Center of Excellence for GRIDs

INTRODUCTION

ABOUT SECURITY TEAM OF PSNC

- Security administration of the PIONIER network infrastructure and PSNC's supercomputing resources
- Performing real-life, large scale penetration tests and security audits of software for third parties (both commercial and educational ones)
- Participation as security consultants in research projects founded by Polish State Committee for Scientific Research, EC and IT vendors
- Extensive knowledge of attack methodologies and techniques
- Continuous security vulnerability research

INTRODUCTION

GOAL OF THE PRESENTATION

- Practical security is based both on knowledge about protection as well as about threats
- Regardless of the very solid security design of Java technology, there have been many problems in Java VM implementations in the past
- So far, no problems have been reported in KVM, which might indicate that its security has been taken on a „as is” basis
- Showing the „real threat” of a given technology is usually the only way to build proper awareness among its users, it also motivates vendors to make a given technology more secure

JAVA 2 MICRO EDITION

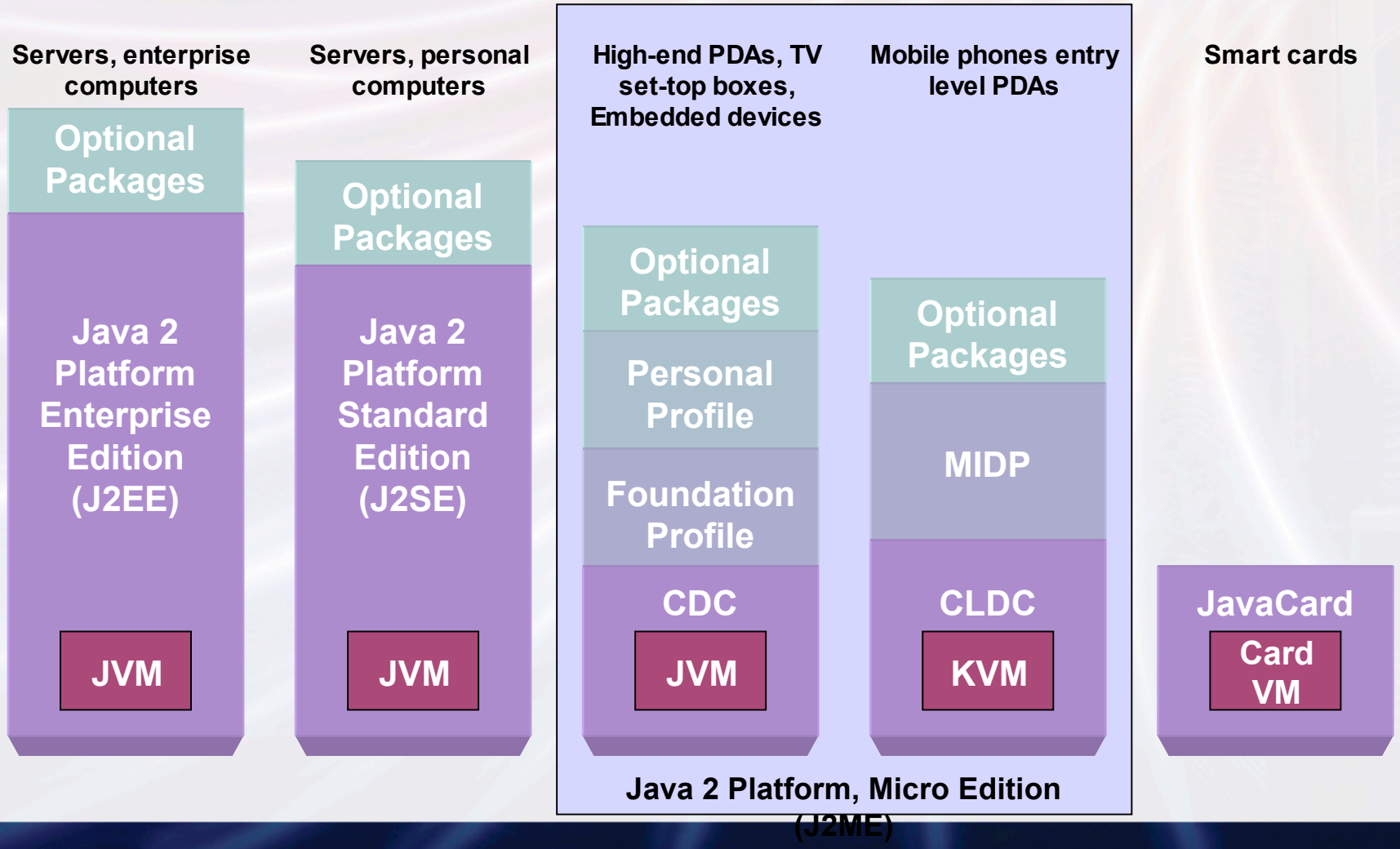
INTRODUCTION

Java 2 Micro Edition (J2ME) has been developed primarily as a technology for the execution of Java applications on small, constrained devices with limited resources (mobile phones, PDAs, TV set-top boxes, in-vehicle telemetry, residential gateways and other embedded devices).

It has been derived from J2SE (Standard Edition) and has all of the characteristics of the Java language.

JAVA 2 MICRO EDITION

WHERE DOES IT FIT IN THE JAVA TECHNOLOGY?



JAVA 2 MICRO EDITION

KEY CONCEPTS

- **Configuration**

  In J2ME jargon, a `configuration' is a set of hardware functionality made available to applications through a JVM. J2ME defines two basic configurations: CDC and CLDC
- **Profile**

  `Profile' is J2ME jargon for the API exposed by the J2ME implementation. There are four basic profiles defined by the J2ME specifications: MIDP, Foundation, Basis, and Personal

JAVA 2 MICRO EDITION

KEY CONCEPTS (2)

- **CDC and CLDC (Connected Device Configuration and Connected Limited Device Configuration)**
  Specifications for the basic JVMs that a J2ME device must support. CDC and CLDC are not really APIs (although CLDC does specify a basic API), but run-time environments. The heart of a given CDC/CLDC implementation is the *Java Virtual Machine.*
- **MIDP (Mobile Information Device Profile)**
  The API exposed by small mobile devices with graphical displays. MIDP is the most widely supported API.

JAVA 2 MICRO EDITION

K-VIRTUAL MACHINE

The K Virtual Machine (KVM), a key feature of the J2ME architecture, is a highly portable Java virtual machine designed from the ground up for small memory, limited-resource, network-connected devices such as cellular phones, pagers, and personal organizers.

KVM is suitable for 16/32-bit RISC/CISC microprocessors with a total memory budget of no more than a few hundred kilobytes (128-256 kB out of which 40-80 kB is needed for the VM itself and the rest is reserved for configuration and profile class libraries)

J2ME SECURITY ARCHITECTURE

SECURITY MODEL

The security model of J2ME is defined at two levels:

- **Low-level (Virtual Machine) security**

  It ensures that the applications running in the virtual machine follow the semantics of the Java programming language, and that an ill-formed or maliciously-encoded class file does not crash or in any other way harm the target device.

  In a standard Java Virtual Machine implementation, low-level security is guaranteed by the *bytecode (class file) verifier.*

J2ME SECURITY ARCHITECTURE

SECURITY MODEL (2)

- **Application level security**
  In CLDC, application-level security is accomplished by using a metaphor of a closed "*sandbox*." An application must run in a closed environment in which the application can access only those libraries that have been defined by the configuration, profiles, and other classes supported by the device. Java applications cannot escape from this sandbox or access any libraries or resources that are not part of the predefined functionality. The sandbox ensures that a malicious or possibly erroneous application cannot gain access to system resources.

J2ME SECURITY ARCHITECTURE

THE SANDBOX MODEL

The sandbox model requires that:

- The downloading, installation, and management of Java applications on the device takes place in such a way that the application programmer cannot modify or bypass the standard class loading mechanisms of the virtual machine.
- A closed, predefined set of Java APIs is available to the application programmer, as defined by CLDC, profiles (such as MIDP) and manufacturer-specific classes.

J2ME SECURITY ARCHITECTURE

THE SANDBOX MODEL (2)

The sandbox model also requires that:

- The set of native functions accessible to the virtual machine is closed, meaning that the application programmer cannot download any new libraries containing native functionality or access any native functions that are not part of the Java libraries provided by CLDC, profiles or manufacturer-specific classes.

J2ME SECURITY ARCHITECTURE

ADDITIONAL SECURITY RESTRICTIONS

A CLDC implementation shall ensure that:

- the application programmer cannot override, modify, or add any classes to these protected system packages,
- the application programmer is not allowed to manipulate the class file lookup order in any way,
- by default, a Java application can load application classes only from its own Java Archive (JAR) file.

J2ME SECURITY ARCHITECTURE

JAVA LANGUAGE SECURITY FEATURES

In Java, security of data is imposed on a language level through the use of access scope identifiers (*private*, *protected*, *public* and *default*) limiting access to classes, field variables and methods.

Java also enforces memory safety (through VM) since security of mobile code can be seen in a category of the secure memory accesses.

J2ME SECURITY ARCHITECTURE

JAVA LANGUAGE SECURITY FEATURES (2)

- Garbage collection
  - memory can be implicitly allocated but not freed
- Type safety
  - strict type checking of instruction operands
  - no pointer arithmetic
- Runtime checks
  - array accesses
  - cast operations
- UTF8 string representation

J2ME SECURITY ARCHITECTURE

DIFFERENCES FROM J2SE

The following Java language features have been eliminated in CLDC because of library changes or security concerns:

- Java Native Interface (JNI)
- User-defined class loaders
- Reflection
- Thread groups and daemon threads
- Finalization
- Weak references
- Errors

J2ME SECURITY ARCHITECTURE

BYTECODE VERIFIER

Bytecode verifier ensures that the bytecodes and other items stored in class files cannot contain illegal instructions, cannot be executed in an illegal order, and cannot contain references to invalid memory locations or memory areas that are outside the Java object memory (the object heap).

Bytecode verifier ensures that Java code is type safe and adheres to the semantics of the Java programming language.

J2ME SECURITY ARCHITECTURE

BYTECODE VERIFIER (2)

Bytecode Verifier checks that:

- code does not forge pointers,
- class file format is OK,
- code does not violate access privileges,
- class definition is correct,
- code does not access one sort of object as if it were another object.

J2ME SECURITY ARCHITECTURE

BYTECODE VERIFIER (3)

Bytecode Verifier guarantees that:

- no stack overflows occur,
- no stack underflows occur,
- all local-variable uses and stores are valid,
- bytecode parameters are all correct,
- object fields accesses (public/private/protected) are legal.

J2ME SECURITY ARCHITECTURE

CLASS FILE VERIFICATION

The existing J2SE bytecode verifier defined in *Java Virtual Machine Specification* is not ideal for small, resource-constrained devices:

- It takes a minimum of 50 kB binary code space, and at least 30-100 kB of dynamic RAM at runtime,
- CPU power needed to perform the iterative dataflow algorithm in the conventional verifier can be substantial.

J2ME SECURITY ARCHITECTURE

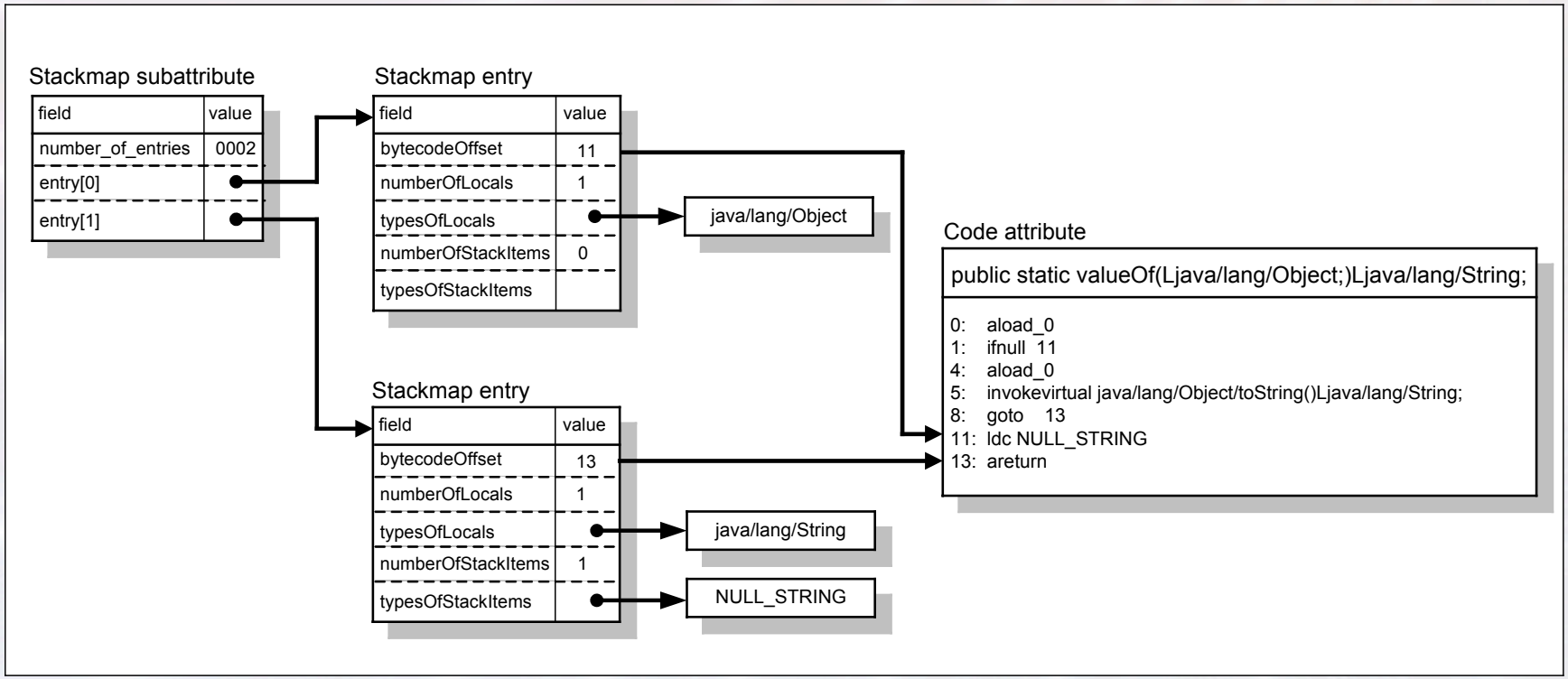
CLASS FILE VERIFICATION (2)

New approach to class file verification based on a two-phase verification process has been implemented in J2ME:

- off-device pre-verification during which J2SE bytecode verifier is used to preverify the bytecodes and special stack map attributes are added into class files to facilitate runtime verification
- runtime verification with stack maps

J2ME SECURITY ARCHITECTURE

STACKMAP ATTRIBUTE



Stackmap attributes describe the state of stack and local variables for code locations that can be reached across several different execution paths (the case for conditional/uncoditional branches, but also exception handlers)

J2ME SECURITY ARCHITECTURE

BYTECODE VERIFICATION ALGORITHM

Bytecode Verification algorithm is based upon data-flow analysis. It is done by modeling the execution of every single bytecode instruction and by inspecting every execution path that can actually occur in a code of a given method.

For each instruction, information about the number of registers used, the stack height and the types of values contained in registers and the stack are maintained (state information).

J2ME SECURITY ARCHITECTURE

BYTECODE VERIFICATION ALGORITHM (2)

Linearly iterate through code and for each instruction:

- Verify instruction operands (types)
- Simulate execution of the instruction
- Compute new state information
- If necessarily, match the derived state with any stack map entries recorded for any successor instructions that do not directly follow the current instruction
- Detect any type incompatibilities

More detailed information about the bytecode verification algorithm can be found in a supplement to the *CLDC Specification - "CLDC Byte Code Typechecker Specification*" by Gilad Bracha, Tim Lindholm, Wei Tao and Frank Yellin

J2ME SECURITY ARCHITECTURE

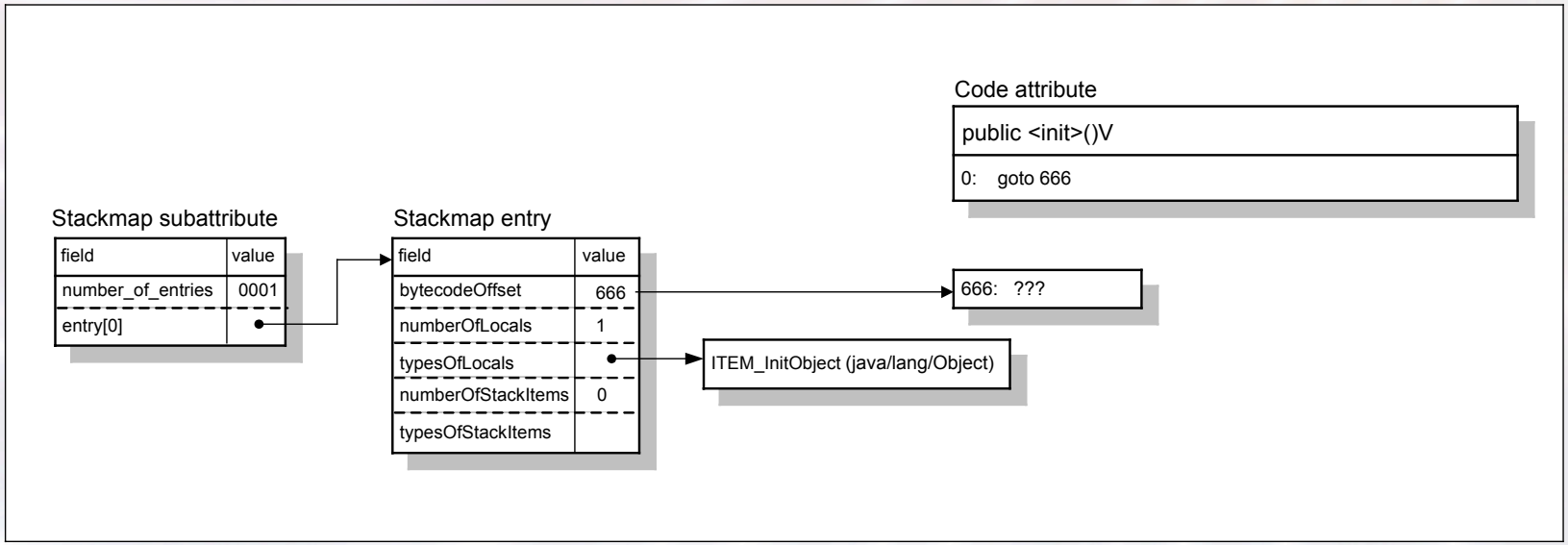
POTENTIAL WEAK POINTS

There are several J2ME architecture components that are critical for its security. This includes:

- Bytecode verifier
- KVM Runtime (execution engine)
- JIT compiler (currently only relevant for Monty VM)
- Core CLDC and MIDP classes
- Vendor specific classes (`com.symbian.*`, `com.sun.*`, `com.nokia.`, etc.)
- Native methods implementation

KVM VULNERABILITIES

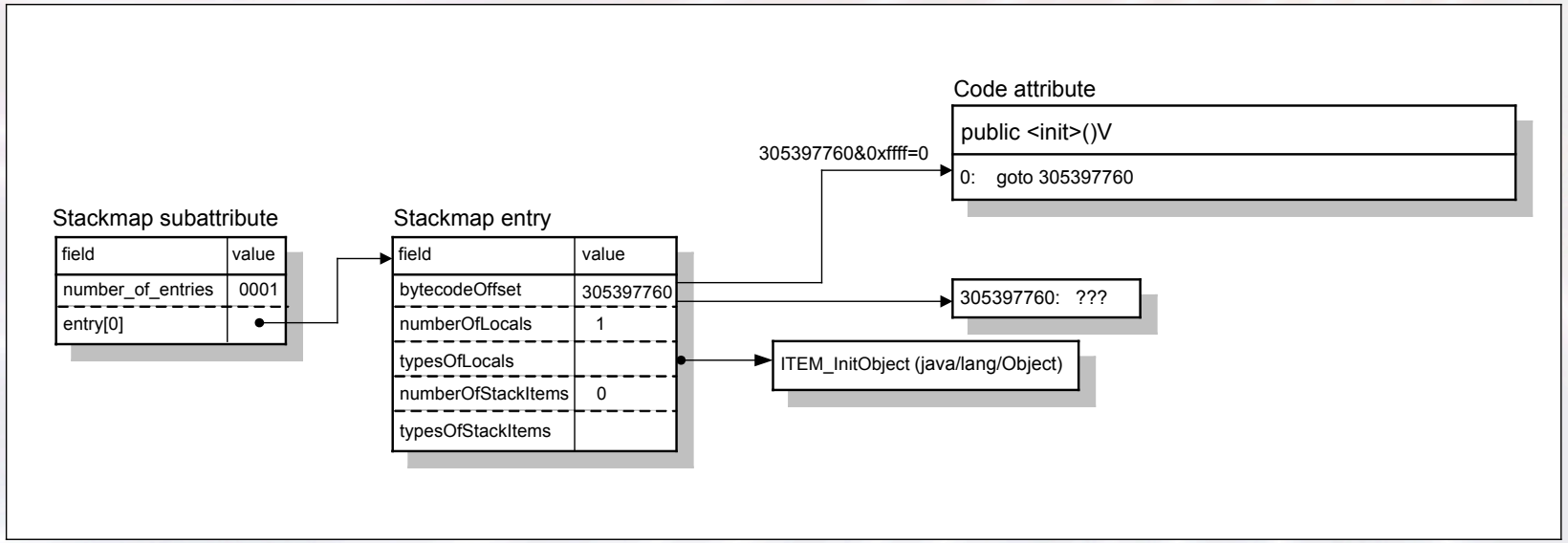
GOTO BYTECODE VERIFIER FLAW



- Bytecode verifier does not check whether the target of the *goto* instruction is within the code limits of the current method
- It only verifies whether there exists a proper *Stackmap* entry corresponding to the target of the *goto* instruction

KVM VULNERABILITIES

GOTO_W BYTECODE VERIFIER FLAW



- Before checking whether the target of the *goto_w* instruction is legitimate, bytecode verifier casts 32 bit offset from the instruction (denoting the target of a jump) to 16 bits (the length of the bytecode offset from a Stackmap entry)
- In a result bytecode verifier follows wrong execution path (jump offset 0)

KVM VULNERABILITIES

EXPLOITATION

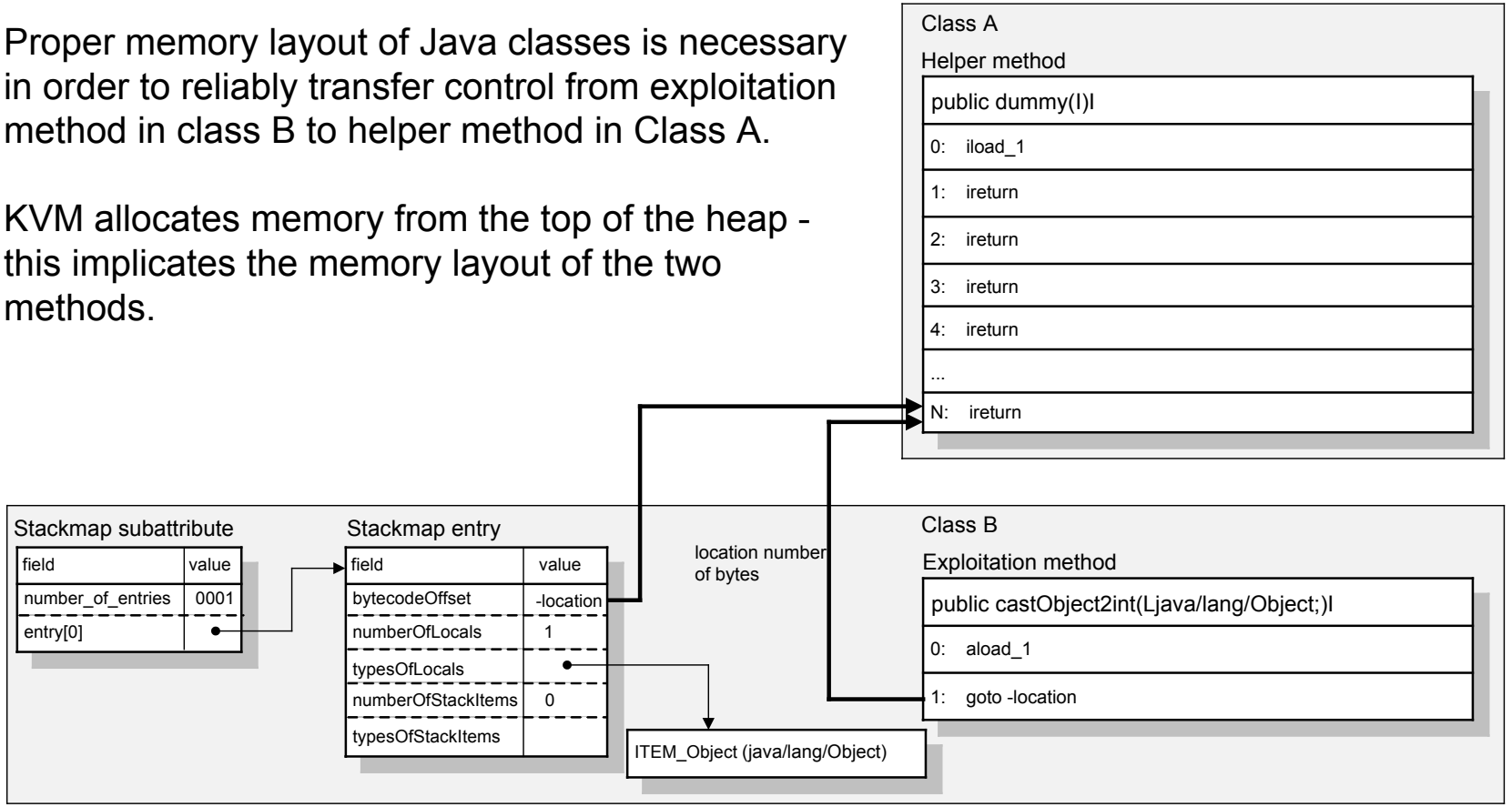
- In a result of both `goto` and `goto_w` bytecode verifier flaws, jumping to arbitrary bytecode locations outside of the current method code can be done
- Since bytecode verifier operates on a single method basis, it is possible to escape the KVM sandbox and to execute Java bytecodes from the unverified execution path
- This opens the whole range of other attack possibilities aiming to circumvent type safety of the Java language in order to get full access to the device memory

KVM VULNERABILITIES

EXPLOITATION (2)

Proper memory layout of Java classes is necessary in order to reliably transfer control from exploitation method in class B to helper method in Class A.

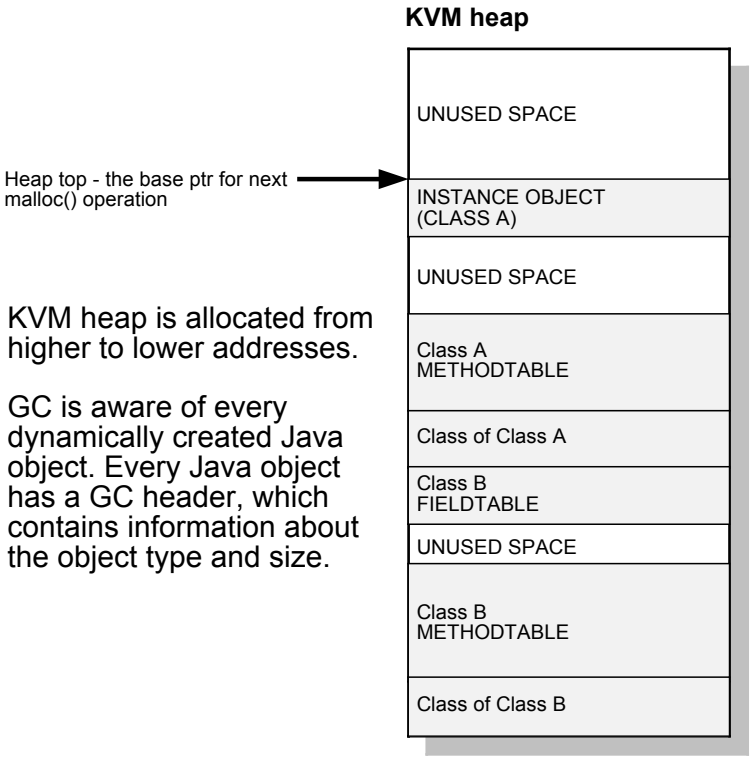
KVM allocates memory from the top of the heap - this implicates the memory layout of the two methods.



KVM VULNERABILITIES

EXPLOITATION RELIABILITY

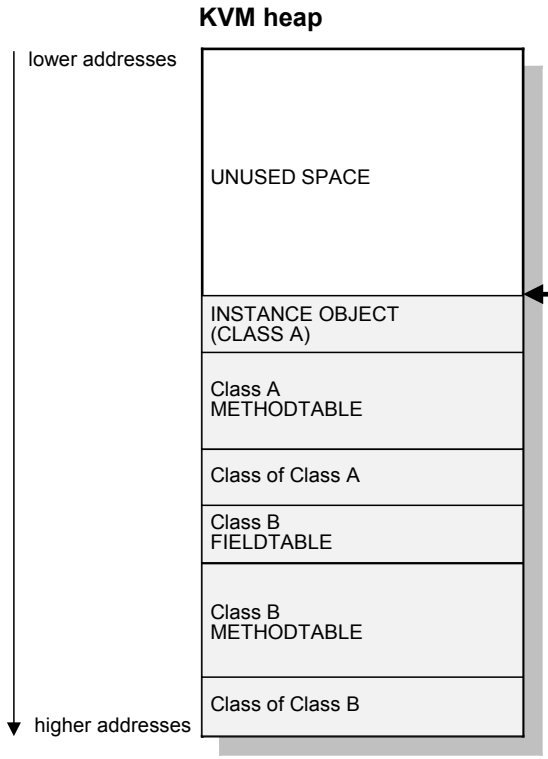
BEFORE GC operation



KVM heap is allocated from higher to lower addresses.

GC is aware of every dynamically created Java object. Every Java object has a GC header, which contains information about the object type and size.

AFTER GC operation



Garbage Collector compacts the memory on user request through System.gc() or upon detecting that there is no free space required for the next Java malloc() operation

GC operation changes the whole layout of the Java heap.

Since addresses of dynamic Java objects constantly change in time it is difficult to rely on them in longer time period. This obviously influences the reliability of the interclass *goto* jump.

KVM VULNERABILITIES

EXPLOITATION RELIABILITY (2)

In order to avoid `IllegalBytecode` exception thrown in a result of executing the `goto` instruction after changing the dynamic memory layout by GC, one can change the body of the exploitation method in such a way, so that there would be proper `ireturn` instruction in its code instead of the `goto` one.

For that purpose, memory safety must be broken first and several interclass `goto` jumps must be made. However, upon proper exploit code implementation this will never trigger the GC operation before patching a given Java method.

KVM VULNERABILITIES

EXPLOITATION RELIABILITY (3)

It is advised to always use the `ireturn` instruction (instead of `areturn`) at the end of Java methods doing casts from arbitrary memory addresses to arbitrary object references in order to avoid the interaction with GC.

KVM VULNERABILITIES

IMPACT

Vendor	Number of phone models with Java (CLDC 1.0 only)
Alcatel	2
Sony Ericsson	13 (excl. PersonalJava)
LG	6
Motorola	21
Nokia	40 (excl. PersonalJava)
Panasonic	5
Samsung	7
Siemens	13

KVM VULNERABILITIES

IMPACT (2)

Only CLDC 1.0 implementations were taken into account while preparing the table from the previous slide (I haven't tested any CLDC 1.1 device). Until vendors confirm that a given device is vulnerable, this table should be treated as presenting only potentially vulnerable devices.

The complete list of J2ME devices can be found at SUN Microsystem's website:

http://jal.sun.com/webapps/device/device

BREAKING JAVA TYPE SAFETY

INTRODUCTION

Because Java is a type safe language, any type conversion between data items of a different type must be done in an implicit way:

- primitive conversion instructions (*i2b, i2c*, *i2d*, *i2f*, *i2l*, *i2s*, *l2i, l2f*, *l2d*, *f2i*, *f2l*, *f2d*, *d2i*, *d2l*, *d2f*),
- *checkcast* instruction,
- *instanceof* instruction.

BREAKING JAVA TYPE SAFETY

TYPE CONFUSION ATTACK

The *type confusion* condition occurs in a result of a flaw in one of the Java Virtual Machine components, which creates the possibility to perform cast operations from one type to any unrelated type in a way that violates the Java type casting rules.

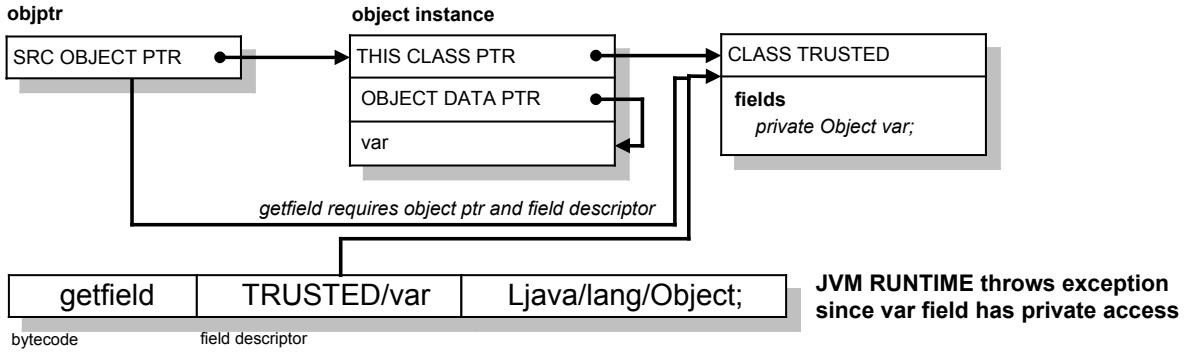
The goal is to perform illegal cast and to access memory region belonging to an object of one type as if it was of some other unrelated type

BREAKING JAVA TYPE SAFETY

TYPE CONFUSION ATTACK (2)

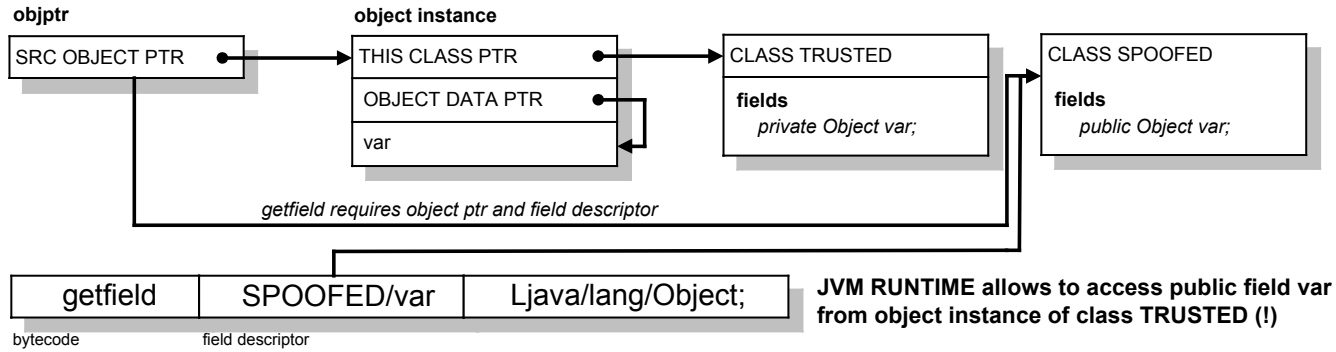
BEFORE TYPE CONFUSION ATTACK

Object o=objptr.var;



AFTER TYPE CONFUSION ATTACK

spoofedptr=BlackBox.cast2SpoofedClass(objptr);
Object o=spoofedptr.var;



BREAKING JAVA TYPE SAFETY

TYPE CONFUSION ATTACK (4)

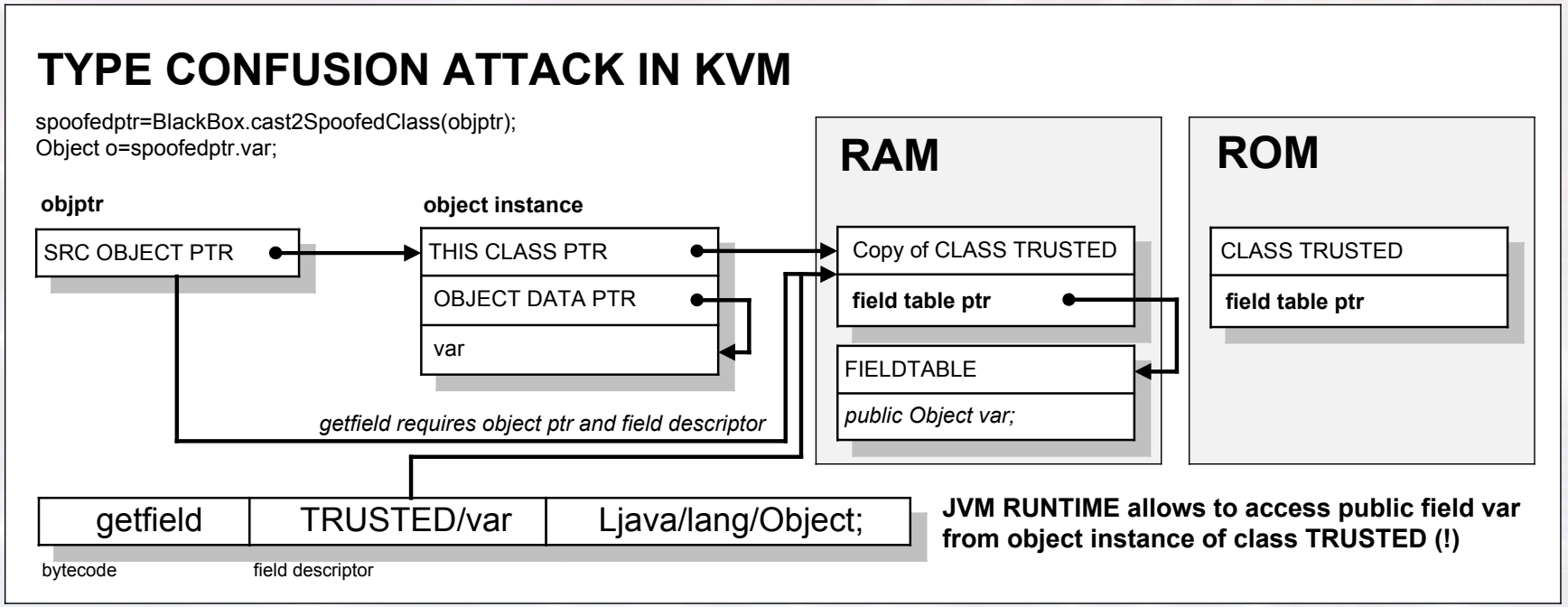
In J2SE *type confusion* attacks are possible since there are no runtime checks done for *getfield/putfield* instructions with regard to the types of their arguments.

They are also possible in J2ME, but need to be modified a bit to reflect different environment and specifically the fact that:

- system classes are in ROM
- bytecodes are interpreted (not compiled into native code)
- Field and methods are looked up dynamically by a hash key computed from a field/method descriptor

BREAKING JAVA TYPE SAFETY

TYPE CONFUSION ATTACK (5)



If not modified, classic *type confusion* attack would not work in KVM, since fieldLookup(thisClassPtr,field/method descriptor) operation would always return NULL for spoofed field descriptors and system classes.

BREAKING MEMORY SAFETY

INTRODUCTION

The goal of breaking memory safety is to obtain unlimited read and write access to the native memory of a host device.

Memory safety can be easily broken with the use of *type confusion* attack.This can be accomplished in at least two ways:

- through field variable
- through table of bytes

BREAKING MEMORY SAFETY

USE OF THE FIELD VARIABLE

```
public Class Spoofed {
 public int value;
}

int spoofptr=BlackBox.cast2SpoofedClass(ADDR-0x0c)
```

`spoofptr.value` can be now read or written what will result in a read or write access to the memory location ADDR

BREAKING MEMORY SAFETY

USE OF THE TABLE OF BYTES

```
mtab=new byte[12];
/* set table size to 2^32-1 */
mtab[8]=mtab[9]=mtab[10]=mtab[11]=(byte)0xff;
/* get addr of fake array obj header */
int base=BlackBox.cast2int(mtab)+0x0c;
/* do the cast */
static int mem[]=cast2arrayOfBytes(base);
```

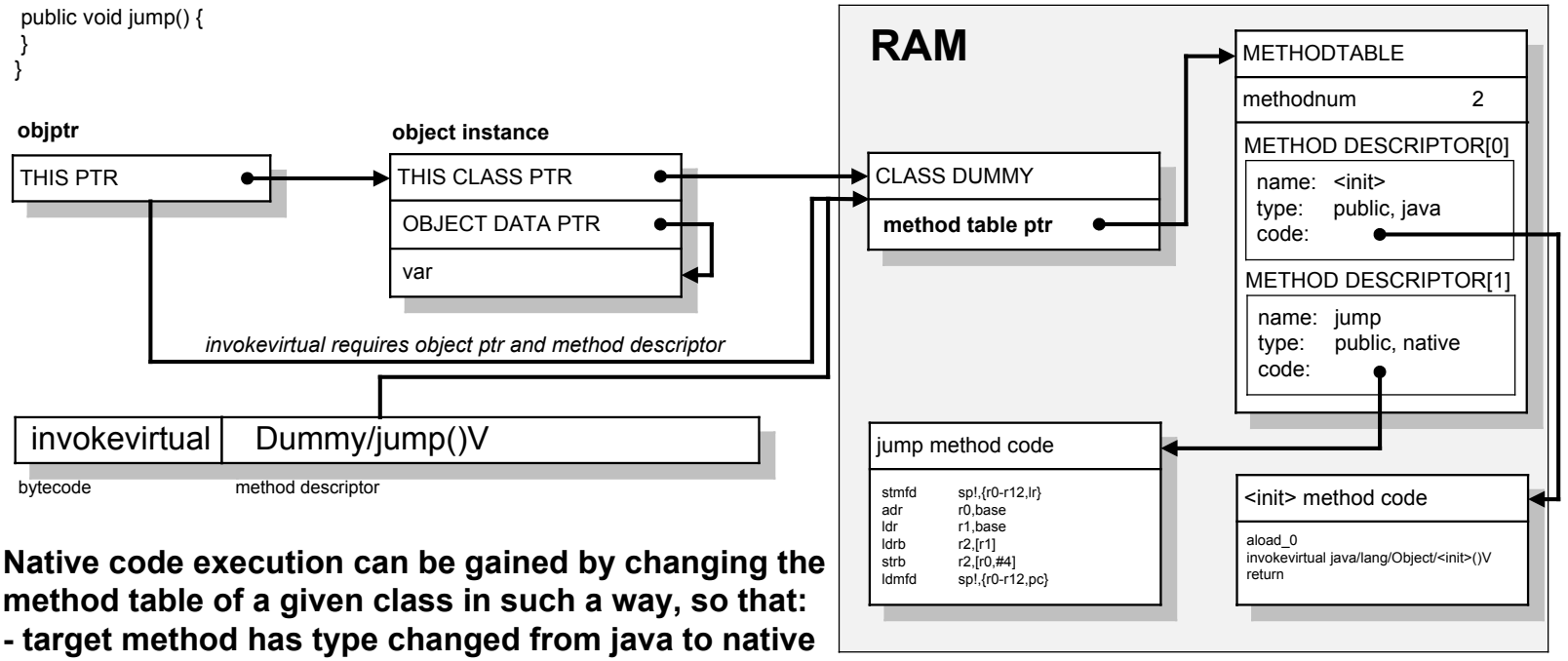
`((int)mem[ADDR-base-0x0c])&0xff` can be now read or written what will result in a read or write access to the memory location ADDR

GAINING CODE EXECUTION ACCESS

HOW TO

SOURCE CODE OF DUMMY CLASS

```
public class Dummy {
 public Dummy() {
 }

 public void jump() {
 }
}
```



Native code execution can be gained by changing the method table of a given class in such a way, so that:
- target method has type changed from java to native
- method address points to user provided code

GAINING CODE EXECUTION ACCESS

RELIABILITY ISSUES

Due to the Garbage Collector behavior, placing assembly instructions to execute in a dynamically allocated Java object (i.e. array of integers) might cause some reliability problems if the code to execute is to be used in long-term, it is the code for the callback or message handler.

In such a case it is much better to use the native OS memory allocation routine for the purpose of obtaining stable memory region.

ACCESSING PROTECTED METHODS

CLASS SPOOFING

Getting access to some protected Java/native methods of the system classes can be accomplished in the following way:

- create the instance of your own Dummy class, with a dummy method and public access (it is important to use the same method name/descriptor as in the spoofed class)
- put the pointer to the method table from the spoofed class to your Dummy class
- invoke a given method from a Dummy class – this will result in the invocation of the corresponding method from the spoofed system class since its method table would be in use

SPOOFING SYSTEM METHODS

CLASS SPOOFING (2)

Class spoofing can be also used for:

- getting access to some protected fields from system classes (in this case field table needs to be spoofed)
- changing definition of some methods from system classes. This considers both Java as well as native code.

Spoofing system classes turns out to be very useful in the case where execution of the system methods needs to be changed (i.e. some security checks/user dialogs need to be bypassed/avoided).

REVERSE ENGINEERING

THE TARGET DEVICE

NOKIA 6310i

- Tri-band world phone - works in three networks on five continents
- Downloadable personal applications via Java- technology
- GPRS (general packet radio service)
- HSCSD (high speed data)
- Bluetooth support
- WAP 1.2.1 browser (over GRPS or CSD)
- Wallet , Customizable and timed profiles, Clock and alarm clock
- Calculator, currency converter, Stopwatch, countdown timer
- Voice commands, voice recorder
- Connectivity options: Bluetooth, infrared, and cable

The so called „closed” DCT4 phone:

- **flashes are available for it only in encrypted form**
- **no way to execute any other user code on it than Java /browse filesystem etc.**

REVERSE ENGINEERING

TASK DEFINITION

To find out what is actually possible to do from within the untrusted midlet application exploiting one of the bytecode verifier flaws in the environment of the Nokia 6310i phone.

Task input:

Java midlet escaping the KVM sandbox and breaking memory safety (can read/write memory and execute ARM code on a device).

REVERSE ENGINEERING

STEP 1: PEEK AND POKE IN MEMORY

By peeking and poking in native device memory it is possible to obtain information about the visible address space. It is also possible to locate many C strings - those which are of special interest here usually begin with „`java/`” or „`com/nokia`”.

These strings can reveal information about what unpublished, vendor specific classes are implemented by a given device:

```
com.nokia.mid.impl.isa.io.protocol.wap.Protocol
com.nokia.mid.impl.isa.jam.Jam
com.nokia.mid.impl.isa.ui.MIDletManager
com.sun.cldc.io.j2me.socket.Protocol
...
```

REVERSE ENGINEERING

STEP 1: PEEK AND POKE IN MEMORY (2)

By peeking and poking in native device memory it is also possible to obtain information about internal Java objects representation and specifically:

- object instance representation
- GC header
- Class structure
- method table
- field table

It’s all about testing different Class definitions (different access, method/field number and types) and creating object instances.

REVERSE ENGINEERING

STEP 2: DISASSEMBLE BYTECODE

By disassembling Java bytecode methods it is possible to find out detailed information about their operation.

Disassembling Java bytecode is especially profitable since Java is a type-safe language and any data access/method invocation must be accompanied in it with a detailed information about the types of accessed variables/arguments passed to the invoked method.

REVERSE ENGINEERING

STEP 3: DISASSEMBLE NATIVE CODE

By disassembling native code, information about some system classes operation can be obtained. This includes J2ME core system classes as well as vendor specific classes.

It's good to start from J2ME core system classes since their descriptor types are known. Knowledge of method arguments number and types is the first step to find out more about the way KVM passes arguments to native code.

Looking at the return address register (LR) can reveal the address of the internal bytecode interpretation loop.

REVERSE ENGINEERING

STEP 3: DISASSEMBLE NATIVE CODE (2)

Disassembling native code can also help:

- Find out how to identify methods/fields in method/field table by name upon having method hash ID (the code of `java.lang.Class.forName()` does it)
- build database of „known subroutines” – this speeds up a process a bit
- find out more about the internal KVM operation as it is usually tightly coupled with the underlying operating system (interprocess communication, permanent memory access, GPRS communication)

REVERSE ENGINEERING

STEP 3: DISASSEMBLE NATIVE CODE (3)

Disassembling native code can also help:

- locate different debugging subroutines:

```
addr    rX,"java_server.c, 538: JAVA_LOAD_REQ"
bl      print_debug
```

- Find out more about the way switch statements are implemented by the compiler (they are required for more advanced code analysis/complete call tree walk up/down):

```
addr    rX,switch_table
lsr     rY,rY,2
ldr     rX,[rX,rY]
mov     lr,pc
bx      rX
```

REVERSE ENGINEERING

STEP 4: SEARCH NATIVE CODE

Searching native code by walking down the whole subroutine call tree (with switch statements resolving) allows to quickly detect some specific code patterns. This specifically includes:

- the use of global variables (i.e. `LDR reg,=value`)
- the use of subroutine calls (i.e. `BL location`)
- debug strings (i.e. `ADDR reg,mem`)

Upon previously gathered knowledge about subroutines and global variables this will usually help identify what a given subroutine deals with.

REVERSE ENGINEERING

STEP 5: EMULATE NATIVE CODE

Manual analysis of even medium size machine level code functions is usually very difficult, tiring and it takes a lot of time.

Emulating native code can be very helpful, especially if it is implemented in such a was so that:

- execution of ARM microprocessor working in THUMB mode is properly emulated – this specifically concerns emulation of **all** ARM THUMB mode instructions and stack operation
- execution of conditional branch instructions automatically changes current code location to the one that would be actually taken in a given context (emulation of ZCNV flags)

REVERSE ENGINEERING

STEP 5: EMULATE NATIVE CODE (CONT.)

- the contents of ARM registers, stack and memory is automatically tracked
- the contents of registers and stack is named apropriately to the origin of a given value
- *step in*, *step over* and *runto* functionality is supported
- all memory accesses are virtualized (a cache of written memory is maintained - no writes are actually done to the device memory, upon reading memory first the cache is looked up for a given addr)

REVERSE ENGINEERING

STEP 5: EMULATE NATIVE CODE (CONT.)

- the database of known subroutines can be maintained, so that descriptive name is shown in disassembly instead of a subroutine addr
- changing between different views can be done easily and in a way depending from a given context (for example, from disassembly view, to memory view if a given instruction references a given address, or from a register view to stack view if a given register contains pointer to the virtualized stack)

It would be very difficult, if not impossible, to actually obtain any interesting results in this work, if code emulation technique would not have be applied.

REVERSE ENGINEERING

OBSTACLES

There are several obstacles that must be fighted along the reverse engineering process:

- Nokia 6310i has very painful size limit for a single midlet JAR archive that can be actually installed onto phone (32 kB is not really much when you plan to write ARM emulator!)
- The phone offers about 160kB of RAM for midlet application, which is not usually enough
- Analyzing machine level code on a 4-5 lines long display is very difficult

REVERSE ENGINEERING

STEP 6: SPEEDING UP THE WORK

By locating subroutines that establish communication over wire with a PC, the whole reverse engineering process can be speeded up, especially if:

- The whole emulation work is done off-device on a PC with the use of a debug Agent Midlet running on device
- Agent Midlet is primarily responsible for reading/writing device memory upon request received over wire from a PC
- PC keeps track of ALL the changes (memory writes, subroutine invocations) made along the emulation of a given native code path

REVERSE ENGINEERING

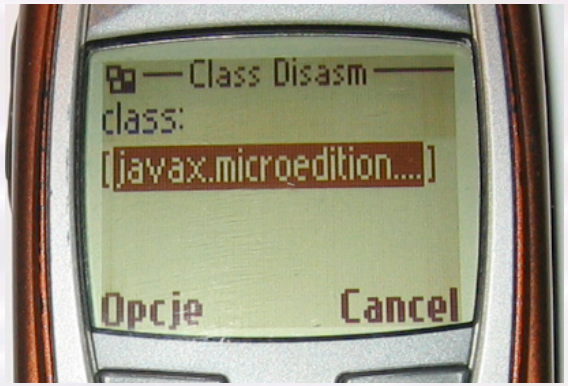
METHOD VIEWER MIDLET

Reverse engineering `java.lang.Class.forName()` can help find out more about the KVM class loading process. It can also help locate subroutines doing class/method/field name hashing:

```
private String get_name(int key) {
  int i,k,addr,val,len,size;
  boolean found=false;
  String name=null;

  key&=0xffff;

  addr=STRING_HASHTABLE_ADDR;
  addr=Mem.read_dword(addr);
  size=Mem.read_dword(addr);

  addr=addr+0x08;
  addr=addr+4*(key%size);
  addr=Mem.read_dword(addr);

  while(!found) {
   if (addr==0) break;
   k=Mem.read_word(addr+0x06);
   if (k==key) {
    found=true;
    break;
   }
   addr=Mem.read_dword(addr);
  }
  if (found) {
   len=Mem.read_word(addr+0x04);
   byte[] tab=new byte[len];
   for(i=0;i<len;i++) {
    tab[i]=(byte)Mem.read_byte(addr+0x08+i);
   }
   name=new String(tab,0,tab.length);
  }
  return name;
}
```

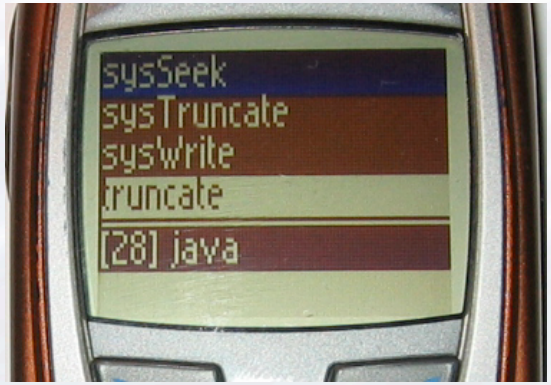
REVERSE ENGINEERING

METHOD VIEWER MIDLET (2)



Main application view

Choosing name of the class of which methods to view



Viewing the methods

REVERSE ENGINEERING

ARMulator MIDLET

Disassembly view

Registers view

Callstack view

Memory view

NOKIA OS

BACKGROUND INFORMATION

- Nokia 6310i runs on ARM microprocessor atop of embedded/RTOS operating system
- There are 0x38 TASKS, which share the same global memory space (RAM and FLASH memory)
- Tasks communicate with each other with the use of message passing IPC mechanism
- JAVA_TASK (tasknum 0x36) runs in `user_level` of ARM microprocessor and is implemented as one native thread
- Input/output peripherals are handled as memory-mapped (also globally visible) with interrupt support

NOKIA OS

BACKGROUND INFORMATION (2)

Structures describing Nokia OS tasks (task name, flags, startaddr, message handler, message queue, etc.) occupy dedicated memory slots (the same for a given SW version).

This is also the case for global variables, which are always located at the same memory addresses for a given SW version.

Nokia phone model and software version can be easily identified by issuing a call to:

```
java.lang.System.getProperty("microedition.platform")
```

NOKIA OS

MEMORY MAP

0x00000000	-	0x0000001F	Interrupt vectors
		0x00000000	Reset Supervisor
		0x00000004	Undefined instruction Undefined
		0x00000008	Software interrupt Supervisor
		0x0000000C	Abort (prefetch) Abort
		0x00000010	Abort (data) Abort
		0x00000014	Reserved Reserved
		0x00000018	IRQ IRQ
		0x0000001C	FIQ FIQ
0x00000020	-	0x000C0000	RAM memory
0x000E0000	-	0x00100000	RAM memory
0x01000000	-	0x017FFFFF	Flash Memory for system code and user data

NOKIA OS

TASKS

[00] MBUS	[13] OBEX	[26] SIM_SERVER
[01] FBUS	[14] VRR	[27] SIMLOCK
[02] RS232	[15] VERSIT	[28] BT_IF_RCV
[03] BT	[16] FILE	[29] TUNE_CONTROL
[04] MDI_RCV	[17] ENERGY	[2a] PND
[05] MDI_SEND	[18] PMM_CLEAN_UP	[2b] UI
[06] STI_RECEIVE	[19] PMM_WRITE_BACK	[2c] CSD_SRV
[07] STI_SEND	[1a] PH	[2d] CSD_NTB
[08] IRDA	[1b] L2	[2e] CSD_FAX
[09] IRDA_MGR	[1c] RR	[2f] SRVS_CKT
[0a] IRDA_PN	[1d] MM	[30] WMLS
[0b] TERMINAL_ADAPTER	[1e] CC	[31] GPRS
[0c] PN	[1f] RM_CONTROL	[32] GPRS_RLC_DLUL
[0d] MONITOR	[20] LCS	[33] GPRS_RLC
[0e] AUDIO	[21] SMS	[34] GPRS_MAC
[0f] MTC_CTRL	[22] GSS_SERVER	[35] GPDS
[10] ACCESSORY	[23] CS_MAIN	[36] JAVA
[11] CORE_HI	[24] SIM_UPL	[37] OS_IDLE
[12] CORE_LO	[25] SIM_12	

NOKIA OS

OBJECTS DIRECTORY

MIDP API provides a mechanism for MIDlets to **persistently** store data and later retrieve it (data is not lost after powering off the phone). This mechanism is available for use through the `javax.microedition.rms.RecordStore` class.

`RecordStore` class does not however contain any native methods, but it references `RecordStoreFile` class which does.

NOKIA OS

OBJECTS DIRECTORY (2)

The following native methods are implemented by the `javax.microedition.rms.RecordStoreFile` class:

```
static native int spaceAvailable();
native int sysClose(int i);
static native int sysCloseDir(int i);
static native int sysDeleteFile(String s);
static native int sysExists(String s);
static native int[] sysGetRMSIds(int i);
native int sysLength(int i);
static native int sysOpenDir();
native int sysOpenFile(String s);
native int sysRead(int i, byte buf[], int j, int k);
static native String sysReadDir(int i);
native int sysSeek(int i, int j);
native int sysTruncate(int i, int j);
native int sysWrite(int i, byte buf[], int j, int k);
```

NOKIA OS

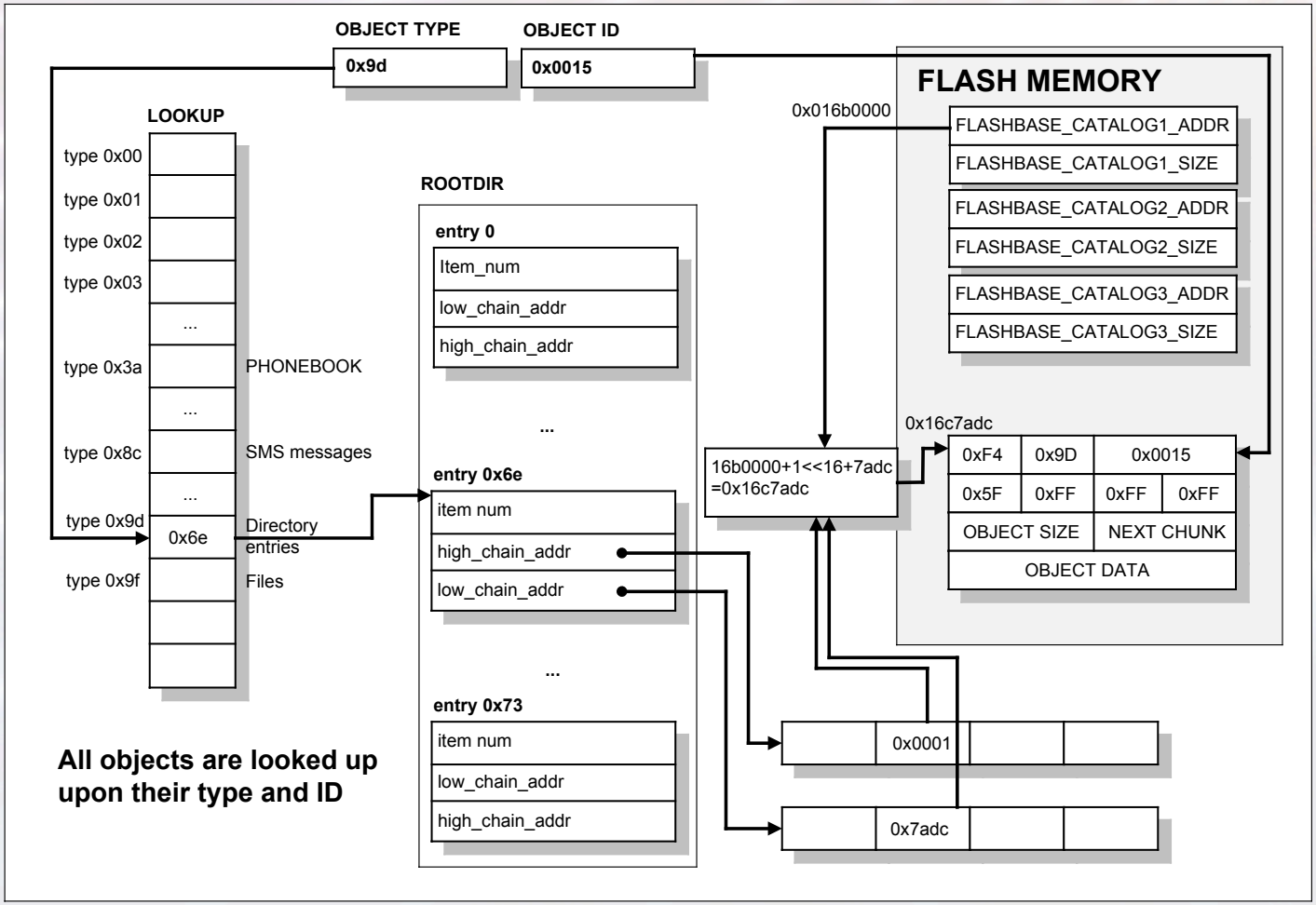
OBJECTS DIRECTORY (3)

Reverse engineering native methods from the `RecordStoreFile` class revealed the way how file system objects (files and directories) are stored in permanent memory (FLASH) of the phone.

More detailed investigation of the discovered mechanism revealed that also some other information pertaining to the phone operation are stored with the use of this mechanism. This includes phonebook, SMS messages, audio records, WAP cache, dialed numbers, etc.

NOKIA OS

OBJECTS DIRECTORY (4)



NOKIA OS

OBJECTS DIRECTORY (5)

File system objects hierarchy (0x9d directory entries):

```
0001 Permanent_memory
            0002 gallery
                        1003 icons
                        1004 photos

            0005 java
                        0006 applications
                                    000a NokiaConverter
                                                000b Cnv_V5_00_en-GB_pl-PL_de-DE.jad
                                                000c Cnv_V5_00_en-GB_pl-PL_de-DE.jad
                                    0010 NokiaWorld clock
                                                0011 Wclk_V5_00_en-GB_pl-PL_de-DE.jad
                                                0012 Wclk_V5_00_en-GB_pl-PL_de-DE.jar
                                                0014 RMS
                                    0015 zupaTest Midlet
                                                0016 test.jad
                                                0017 test.jar
                                                0018 RMS
                        0007 games
                                    000d NokiaRacket
                                                000e Rack_V5_00_en-GB_pl-PL_de-DE.jad
                                                000f Rack_V5_00_en-GB_pl-PL_de-DE.jar
                                                0013 RMS
```

NOKIA OS

INTERPROCESS COMMUNICATION

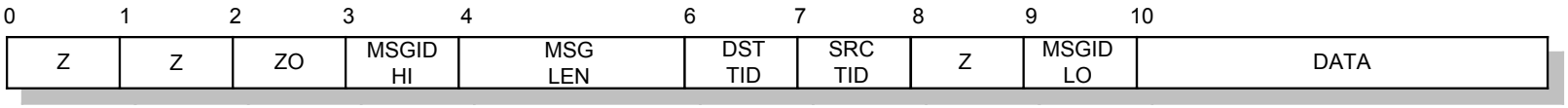
Reverse engineering native methods from the `com.sun.cldc.io.j2me.socket.Protocol` class revealed information about native Nokia OS (system calls), processes (process control structures, message handlers, message buffers) and interprocess communication (the way messages are sent, callbacks are registered).

All of that was possible due to the fact that `open0` method of the `socket` class sends a message (FBUS packet with command 0x83) to the FBUS_TASK and waits for a response.

NOKIA OS

INTERPROCESS COMMUNICATION (2)

Format of messages sent between tasks:



MSGID HI	- high byte of message ID
MSGID LO	- low byte of message ID
MSG LEN	- length of message (counted from DST TID field)
DST TID	- destination task ID
SRC TID	- source task ID
DATA	- message data
Z	- the value of zero was usually observed at this field
ZO	- the value of zero or one were usually observed at this field

NOKIA OS

INTERPROCESS COMMUNICATION (3)

It is possible to write a message tracing application, that simply intercepts message handling routine for a given process and sends messages received to it over wire to the PC.

This can be done from a midlet application since RAM memory is globally shared among native Nokia OS processes.

By doing so, **lots of new information** about the phone internal operation can be obtained.

NOKIA OS

INTERPROCESS COMMUNICATION (4)

It is also possible to install user resident code on the phone, that would intercept (or influence) communication occuring between native Nokia OS processes. Exiting Java application would not end such code provided that:

- proper message handler is intercepted (there are tasks, that constantly send messages to some other tasks, GSM network info is one of the examples)
- native malloc is used for storing user message handler (java malloc cannot be used since JAVA_TASK frees all allocated memory upon midlet's exit)

NOKIA OS

INTERPROCESS COMMUNICATION (5)

A list of processes that can have their message handlers directly intercepted:

```
MONITOR, AUDIO, ACCESSORY, CORE_HI, CORE_LO, OBEX, VRR, VERSIT,
FILE, ENERGY, PH, GSS_SERVER, CS_MAIN, SIM_SERVER, SIMLOCK,
TUNECONTROL, PND, UI, CSD_SRV, SRVSCKT, WMLS, GPRS, GPRS_RLC_DLUL,
GPRS_RLC, GPRS_MAC, GPDS, JAVA
```

Messages sent to other tasks, like for example SMS Server can be also intercepted (it is however not so easy to locate their message handling routines since these are different process types).

NOKIA OS

FLASH MEMORY

Although, Java application can persistently store data in FLASH memory through the `RecordStore` class, reverse engineering it didn't actually reveal any code doing FLASH writes.

The actual FLASH writing is done upon Java application exit in the `com.nokia.mid.impl.isa.ui.MIDletManager` class (its `s_exitInternal` method).

RMS buffers are simply cached in RAM during midlet execution and flushed at its exit (internal `RMSFlushBuffers` subroutine is called for that purpose).

NOKIA OS

FLASH MEMORY (2)

There is a FLASH object in RAM which can be used for the purpose of writing FLASH memory. It contains flash base address and addresses to several subroutines that issue commands to FLASH controller :

- program command
- sector erase, erase suspend, erase resume
- sector lock/unlock

NOKIA OS

FLASH MEMORY (3)

Writing FLASH memory can be also done through direct programming the FLASH controller.

Programming FLASH memory is a four-bus-cycle operation. The program command sequence is initiated by writing two unlock write cycles, followed by the program set-up command. The program address and data are written next, which in turn initiate the FLASH write operation.

Programming flash can only change bits „1” to „0”. In order to set bits to „1”, the whole sector must be first erased.

MALICIOUS MIDLETS

STEALING DATA

By accessing the Nokia OS object directory malicious midlets can access some sensitive data, like phone contact info, SMS messages or dialed numbers.

It is also very probable that some highly sensitive data (PIN, IMEI, PUK) can be also stolen from the phone (this haven't been verified though).

Stolen data can be further sent to some arbitrary Internet address with the use of GPRS/WAP connection.

MALICIOUS MIDLETS

CONNECTING TO THE INTERNET

By default, user is warned when untrusted midlet tries to access Internet. He/she can then decide whether to allow for such an access or not.

It is possible to bypass this mechanism in the following way:

- set up fake JAD file directory entry in permanent memory (this entry contains information about the midlet's Internet access level), `RecordStore` class can be used to write to FLASH, and direct call to internal `RMSFlushBuffers` native subroutine will make the changes immediately visible

MALICIOUS MIDLETS

CONNECTING TO THE INTERNET (CONT.)

- change objects directory in RAM, so that it points to the fake JAD directory entry with Internet access level set to 2 („don't ask - access allowed")
- establish HTTP connection to the Internet
- restore original structure of the objects directory in RAM, so that original JAD directory entry is in place

MALICIOUS MIDLETS

CHANGING PERMANENT MEMORY

The ability to write permanent memory of the phone is the most dangerous action that a given malicious midlet can take. This is particularly caused by the fact that it allows for:

- the change of phone software and potential backdoor installation (this needs more investigation since there might exist some checksums protecting against such a change of phone's code)
- making the phone unusable by erasing the whole FLASH memory (it's only a matter of issuing a chip erase command which takes just 6 write operations)

MALICIOUS MIDLETS

SENDING SMS MESSAGES

The possibility to communicate with other tasks in the phone allows to make use of their rich services.

Malicious midlet can silently send SMS messages (text, picture, etc.) to arbitrary phone numbers with the use of arbitrary SMS centers and arbitrary message content.

SMS sending can be done with the use of properly formatted message (MSG ID 0x0202) sent to MDI_SEND task (TASK ID 0x5d).

MALICIOUS MIDLETS

INTERFERING WITH IPC

The possibility to interfere with interprocess communication actually allows for the interference with user actions taken on the phone.

Malicious midlet can block or change messages sent by a given process in response to user actions. This can for example result in an SMS sniffing application (SMS messages sent by user are also sent to some other number).

It is highly probably that arbitrary phone calls can be also established through IPC (communication with GSMCallServer).

FINAL REMARKS

FUTURE THREATS

- The fact that there are more users of mobile devices than PC's makes it very attractive target for attackers and worm writers
- It should be expected that **remote vulnerabilities** for mobile devices will be published within **next 6 months**
- Vendors and antivirus industry are not prepared for this kind of threats (there are no means to protect users of the so called „closed" mobile devices against malicious code)
- Open platforms (PalmOS, Symbian OS, Windows CE) seem to be easier to protect, but they are also at the most risk

FINAL REMARKS

ABOUT THIS WORK

- All of the KVM attack methods/malicious midlet examples presented in this work have been verified in practice
- Detailed information about the addresses of some native methods, system calls, global variables and Nokia OS/KVM structures (fields, offsets) has been intentionally removed from this presentation
- This also considers the way in which to interoperate with native Nokia OS/KVM code
- Research paper with all the details including some additional material that didn't fit into this 90min talk will be published in a couple of months

FINAL REMARKS

VALUABLE RESOURCES

- *The Java™ Virtual Machine Specification*, Tim Lindholm, Frank Yellin
- *Connected Limited Device Configuration Specification ver. 1.0/1.1,* SUN Microsystems
- *CLDC Byte Code Typechecker Specification*, Gilad Bracha, Tim Lindholm, Wei Tao and Frank Yellin
- *J2ME Building Blocks for Mobile Devices*, *White Paper on KVM and the Connected, Limited Device Configuration (CLDC)*, SUN Microsystems
- KVM Porting Guide, SUN Microsystems

THANK YOU FOR YOUR ATTENTION!

In case of any questions or comments, feel free to contact me at the following address:

adam.gowdiak@man.poznan.pl