The ABAP Underverse

Risky ABAP to Kernel communication and ABAP-tunneled buffer overflows

Andreas Wiegenstein

Andreas.Wiegenstein@virtualforge.com

BlackHat Briefings 2011, Barcelona

Abstract

In most organizations, the efforts to protect SAP systems focus on Roles, Authorizations, Segregation of Duties, Encryption, Password Policies, Firewalls, Network Infrastructure and Patch Management.

The fact, that *application security* needs to be addressed as well has not yet reached the CxOs and development units of most companies.

As a result, most SAP customers neither know that their business security depends on secure code, nor do they invest any effort to write secure SAP custom applications.

The business risk introduced by insecure code is very high. And companies that outsource (at least part of) their development, may have an even higher risk.

There are various security risks in ABAP. Some are known from other languages, many are special to ABAP. This paper focuses on vulnerabilities special to ABAP, more precisely on risks related to the usage of SAP Kernel Calls. It also describes some observations how the ABAP Runtime executes the ABAP byte code.

Table of contents

SAP Security Fundamentals

"It must be green."

According to SAP "The best-run businesses run SAP". Following that logic, the security of the best-run companies in the world depends on the security of their SAP system.

But business security is more than keeping the hackers out. Business Security means complying with legal requirements like SOX or data protection laws. Business Security means protecting the company's assets from industrial espionage. Business Security is vital for passing audits.

For an SAP customer the security goals are simple: every employee / user must be able to execute the business tasks he/she is assigned to. No more. No less. Since the SAP ERP solution already contains most business modules companies need, most companies focus on assuring the secure usage of those business modules.

Therefore, practically all security measures SAP customers implement today focus on protecting the SAP standard installation:

- Secure Configuration & System / Service Hardening
- Secure Network Infrastructure
- Identity Management
- Password Policies
- Assigning proper Roles & Authorizations, considering Segregation of Duties
- Applying new (security) Patches

Of course, protection of the integrity of the SAP standard features is very important, since companies need to be able to rely on SAP mechanisms like:

- Access restrictions to business logic (to enforce business roles)
- Access restrictions to sensitive Data (for legal reasons, e.g. data protection laws)
- Integrity of Change Documents (for financial compliance)
- Audit Logs (for public accountants)

While these efforts are imperative and effectively protect the SAP standard, they do not necessarily result in a fully secure SAP system. *Because they do not address custom code*.

The moment companies add *custom code* to their SAP system, they might accidentally introduce *backdoors* that bypass the (effectively protected) security features of the SAP standard.

Introduction to ABAP Security

"I'm a doctor, not an ABAP developer!"

ABAP is a proprietary business programming language that provides several integrated security mechanisms to protect the design and integrity of SAP standard mechanisms. Since ABAP is proprietary, its specification and inner workings are not (fully) documented and available to the public.

All ABAP applications are available as source code (stored in the SAP database). In an ECC 6.0 system, there are approximately 155 Mio lines of standard ABAP code. When a given ABAP program is first executed on a system, the compiler creates the corresponding byte code (stored in the SAP database). This byte code is platform independent.

But not only the byte code is platform independent. Also the database access is designed to be independent of the (supported) database in use. Therefore, database access in ABAP is done via a special SQL abstraction layer created by SAP: Open SQL. Open SQL is an intermediate layer between ABAP and the actual native database commands. Its commands are integrated in the ABAP language. Open SQL also enforces certain restrictions on database access. For example only a few defined database commands are allowed (SELECT, UPDATE, DELETE, …) and access to client-dependent tables is automatically enforced. Also, database queries can be automatically logged (if this is configured).

Classic ABAP applications (called "transactions") are executed through a proprietary (fat) client called SAP GUI. The communication between SAP GUI and the server is based on a proprietary protocol called DIAG. "Modern" ABAP applications (Business Server Pages and Web Dynpro ABAP) can be executed with a standard Web browser.

A very important aspect in ABAP is that ABAP runs with super-user privileges. ABAP code has *complete* access to the SAP system, the SAP database and the operating system (of the application server). Therefore, all ABAP code that performs critical actions must first determine if the current user has the required privileges. This is done in most cases by performing an explicit authorization check in the code. ABAP therefore ships with a built-in authentication and a highly complex but fine granular authorization mechanism.
SAP's authentication standard already provides comprehensive mechanisms for log in, password policies and policy enforcement (e.g. an account lock out in case of too many false login attempts). SAP's authorization standard provides a comprehensive set of tools for authorization management. However, since the ABAP language has an explicit authorization model, all ABAP programs must explicitly check the current user's authorization in order to prevent privilege escalation. This is done with the command AUTHORITY-CHECK. Any ABAP code without explicit (and proper) AUTHORITY-CHECKs in the code will execute its commands, even if the user has no proper authorization.

Another important security mechanism in the SAP standard is client separation. Client separation is a concept that allows for hosting business data (and user accounts) of multiple different organizations / departments ("clients") on the same SAP system but in complete isolation. This means that users of client "A" can't access any business data of client "B". The ABAP Runtime implicitly adds a WHERE condition to every relevant Open SQL statement in order to enforce client separation.

A few Notes on ABAP's attack surface: most ABAP programs (in the general sense) can be executed via SAP GUI, many via Remote Function Call (RFC), many via Web browser, many via Web Services. All access requires a valid SAP user, either explicit (login with own credentials) or implicit (silent login with a "technical" user). Implicit authentication is done by some Web applications in an anonymous access scenario where users do not have an account.

Some ABAP routines (called function modules) can be configured to be remote-accessible (called "RFC-enabled"). All such RFC-enabled function modules can be directly called from every user with network access to the RFC port of the SAP server. The only pre-requisite for this is a user on the system and the privilege to access the given RFC function module, as defined by authorization object S_RFC. Note that there are more than 33.000 RFC-enabled function modules available in a standard SAP ECC 6.0 system. A granular authorization concept that assigns each user the correct authorization for each function module would be an enormous effort due to its complexity. Therefore, most SAP customers assign their key users S_RFC "*" authorization, which means, that these key users can remotely access all RFC-enabled function modules.

It is best practice to protect all RFC-enabled function modules with an explicit authority check.

Enter the ABAP Runtime

"My God, it's full of stars!"

The ABAP Runtime is the component in the SAP Kernel that processes the ABAP byte code. If an ABAP module is called for which no (current) byte code exists, the corresponding source code is first compiled into byte code and then the byte code is executed. There is also an explicit command in ABAP that generates the corresponding byte code: GENERATE REPORT. In theory, all source code and all byte code executed by the ABAP Runtime are persisted in the database.

This behavior is shown in Figure 1.

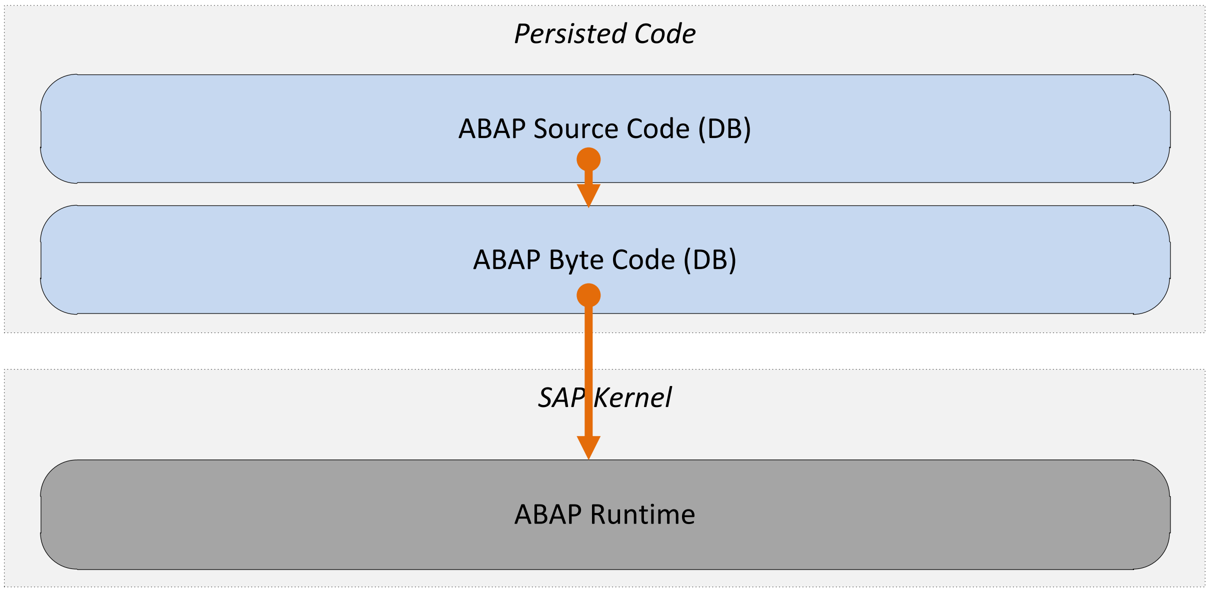

Figure 1: The ABAP Runtime

There are, however, several exceptions to this behavior: Dynamic ABAP Commands, Dynamic ABAP Code Creation, Kernel Communication and Kernel Hooks.

Dynamic ABAP Commands

First, not all ABAP Source code is stored in the database. This is because of SAP's concept of dynamic ABAP programming. Dynamic ABAP programming introduces commands that are not entirely compiled into byte code. Although most of the ABAP statement is turned into static byte code, some sections of it are dynamically computed based on the contents of variables the statement processes. This concept is somewhat similar to Prepared Statements in SQL. However, in ABAP the variable that is dynamically used can contain commands. In the following, I explain this concept with two examples (out of many).

1. Dynamic usage of ASSIGN

The first code snippet assigns the contents of variable lv_tmp (which is user input) to the field symbol <fs>. Assigning here means that <fs> contains a reference (pointer) to lv_tmp, rather than a copy of the contents of lv_tmp. This is static coding.

```
REPORT ZSTATIC_ASSIGN.

DATA lv_secret TYPE string.

FIELD-SYMBOLS <fs> TYPE ANY.

PARAMETERS lv_tmp TYPE string DEFAULT 'Bishop'.


ASSIGN lv_tmp TO <fs>.
```

The next code snippet uses a dynamic variant of the ASSIGN command. Note the parenthesis around variable lv_tmp. In this case, <fs> does not receive a reference to variable lv_tmp but to the variable represented by the value of variable lv_tmp. Therefore, if the default value is not changed by a user, the ASSIGN command would assign a reference of variable lv_share to the field symbol <fs>. If a user e.g. enters lv_secret instead, than the contents of lv_secret would be referenced by <fs>.

```
REPORT ZDYNAMIC_ASSIGN.

DATA lv_secret TYPE string VALUE 'ZFT'.

DATA lv_share TYPE string VALUE 'Bishop'.

FIELD-SYMBOLS <fs> TYPE ANY.

PARAMETERS lv_tmp TYPE string DEFAULT 'lv_share'.


ASSIGN (lv_tmp) TO <fs>.
```

2. Dynamic usage of a WHERE clause in Open SQL

The first code snippet shows code that uses an Open SQL command in order to delete a dataset in a table of the SAP database. The WHERE condition is explicit and identifies the dataset in table USR02 that belongs to the currently logged-on user.

```
REPORT ZSTATIC_DELETE.

DELETE FROM usr02 WHERE bname = sy-uname.
```

The second code snippet uses a dynamic WHERE condition, which is based on user-input. The default result is the same. However, since the WHERE condition is input, it could also be changed in order to affect more than one dataset, e.g. by feeding the input `'bname = LIKE `%`'`, which would affect all datasets in the given table. If malicious input reaches a dynamic WHERE clause, then this results in an Open SQL Injection vulnerability.

```
REPORT ZDYNAMIC_DELETE.

PARAMETERS lv_tmp TYPE string DEFAULT 'bname = sy-uname'.

DELETE FROM usr02 WHERE (lv_tmp).
```

These examples show that ABAP commands may contain dynamic portions that are evaluated by the ABAP Runtime each time a statement is executed. In this case, neither the entire source code nor the effectively executed byte code is persisted. As a result, ABAP commands may be executed in unwanted and potentially dangerous ways. More importantly, such unwanted behavior is not obvious to SAP administrators and will leave almost no traces in the SAP logs.

Risk & Relevance: HIGH. The behavior of dynamic ABAP statements depends on (external) input. Malicious input may result in unwanted side effects like Open SQL Injection. Also, dynamic ABAP is more difficult to audit than static ABAP code.

Usage of dynamic ABAP commands is shown in Figure 2.

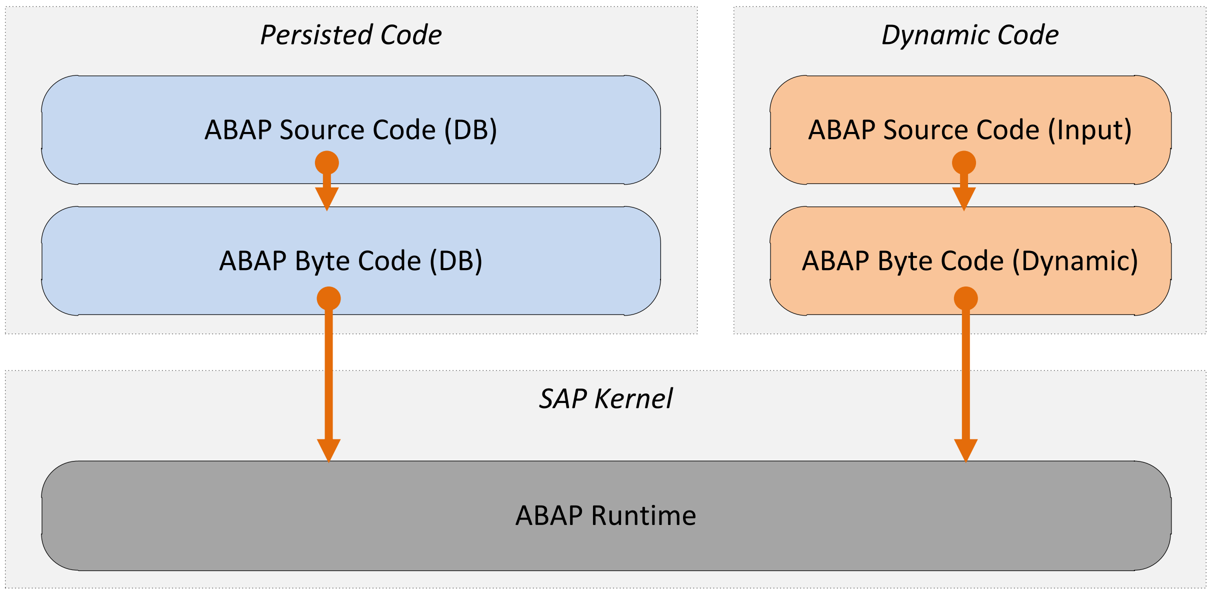

Figure 2: Dynamic Commands in the ABAP Runtime

Dynamic ABAP Code Creation

The SAP standard way to write ABAP code is to use transaction SE80 (transaction is the SAP term for a dialog program). Users need special developer authorizations in order to use this transaction. Reading source code requires authorizations, changing or adding source code requires authorizations, running code requires authorizations and debugging code also requires authorizations. However, on a productive SAP system no one should have the privilege to change ABAP code. But again, authorizations in ABAP are explicitly checked in the code. If the authority check is missing, users may read / create / execute / debug code without proper privileges.

ABAP provides two commands that can dynamically create new ABAP code based on input: INSERT REPORT and GENERATE SUBROUNTINE POOL.

INSERT REPORT turns dynamic source code (input) into persisted source code, as shown in Figure 3. If this command is present in a custom ABAP program without proper authorization checks, it could be used as a back door to create / change ABAP programs permanently on a productive system.

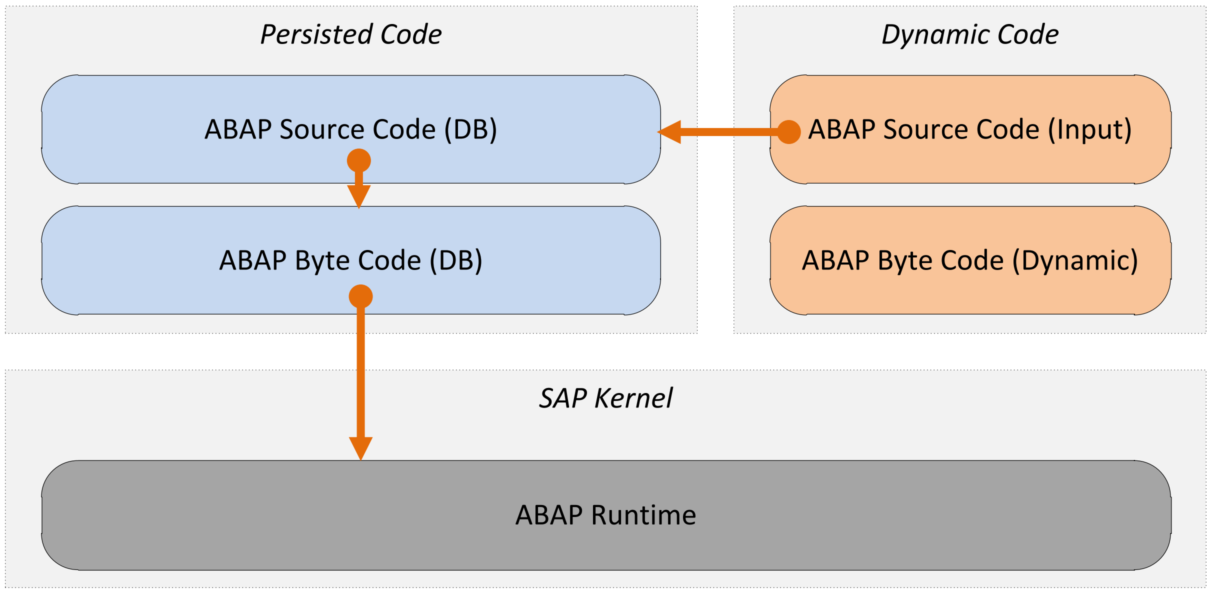

Figure 3: Effect of INSERT REPORT

GENERATE SUBROUTINE POOL turns dynamic source code (input) into executable dynamic byte code. One other nice effect of this command is that the generated program is outside the customer namespace, as is starts with '%_'. GENERATE SUBROUTINE POOL (in combination with PERFORM) effectively enables developers to write programs that create arbitrary ABAP code on the fly, turn it into byte code and execute it without ever touching the database. If malicious code is created this way, there will be virtually no traces left to investigate. This concept is shown in Figure 4.

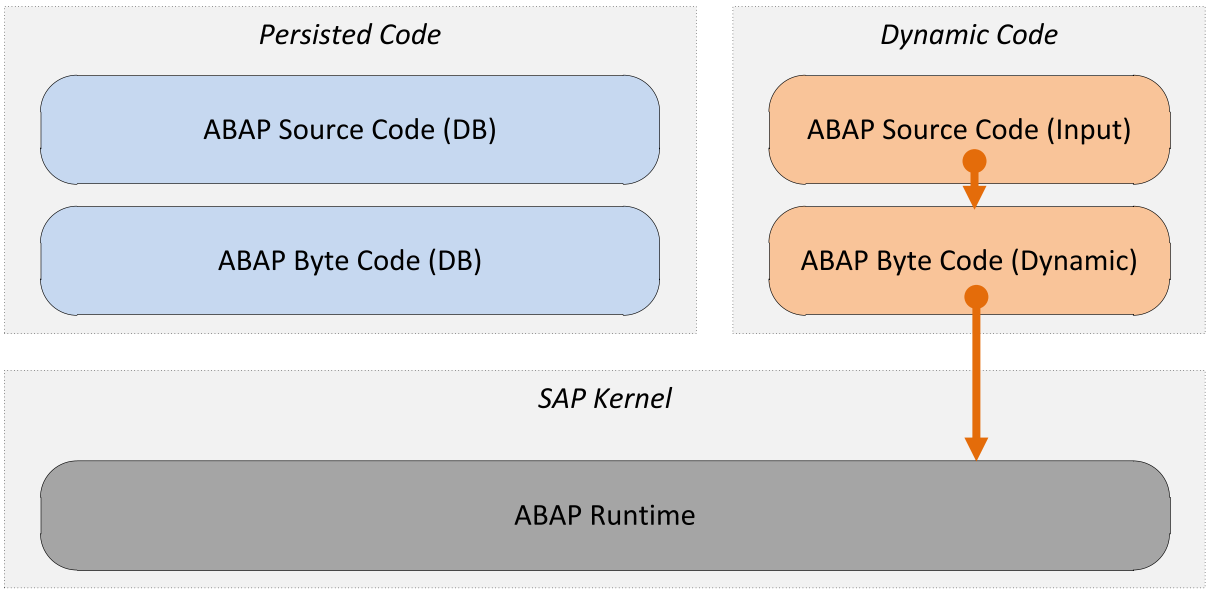

Figure 4: Effect of GENERATE SUBROUTINE POOL

These examples show that two particularly dangerous ABAP commands can bypass the (secure) SAP standard way to create and execute code. They also (again) show that there are ways to create and execute ABAP code that leave no traces in the SAP system.

Risk & Relevance: VERY HIGH. If dynamically created ABAP code contains input, an attacker can easily compromise the security of the entire SAP system by injecting ABAP commands. Such a vulnerability will give an attacker full control over the database, including financial data, HR data and - of course - the user authorizations, which are in turn stored in the database.

Kernel Communication

Although the ABAP language contains more than 300 commands, there are situations where additional functionality is required. In such cases, the ABAP (standard) code explicitly calls (undocumented) methods in the SAP Kernel. These methods can't be reached by other means and were written as helpers for the ABAP Runtime.

There are three different types of ABAP to Kernel communication: Kernel Calls, System Calls and Kernel Methods. Each is bound to a particular ABAP command.

1. Kernel Calls

By means of a Kernel Call, (custom) ABAP code can directly invoke special C/C++ functions in the SAP Kernel and interchange data with them. Kernels Calls are performed by the command CALL *cfunc*. The call is followed by a list of NAME/VALUE pairs. The following code example shows, how SAP profile parameter 'SAPDBHOST' is read via a Kernel Call into ABAP variable lv_dbhost.

```
REPORT ZKERNEL_CALL.
DATA lv_dbhost LIKE msxxlist-host.
CALL 'C_SAPGPARAM' ID 'NAME' FIELD 'SAPDBHOST'
                   ID 'VALUE' FIELD lv_dbhost.
```

There is no formal way to identify which VALUEs are sent to the kernel and which are retrieved. This depends on the individual Kernel Call and has to be analyzed be the developer invoking the call. Also, there is no (comprehensive) documentation which parameters can actually be interchanged per Kernel Call.

The SAP documentation clearly states that Kernel Calls should not be used, since they are meant for internal use only. It also (incorrectly) states that the statement cannot be used in application programs. However, Kernels Calls are one of the rare instances in ABAP where an implicit authority check is performed. To be more precise: this implicit authority check is performed only for the Kernel Call statement CALL 'SYSTEM'. All other Kernel Calls are always executed by the kernel, even if they perform critical actions.

Historically, Kernel Calls where the first technical means to actively interchange data with the SAP kernel. They are officially deprecated (but still work) and were to be replaced by System Calls.

Still, there are currently still more than 370 different Kernel Calls in active use in an SAP ECC 6.0 system, with more than 10.000 call instances.

2. System Calls

By means of a System Call, (custom) ABAP code can directly invoke special C/C++ functions in the SAP Kernel and interchange data with them. Technically, System Calls are very close to Kernel Calls. However their interface is not as uniform as in Kernel Calls. System Calls are performed by the command SYSTEM-CALL. The SAP documentation very clearly (with 6 exclamation marks) states that System Calls are reserved for internal use and are to be used in SAP Basis development only. This means that even other development units inside SAP should not use them! (!!!!!)

The following code snippet reads an HTTP request parameter ('chevron_V') from the HTTP request into the ABAP variable lv_value. This example illustrates that System Calls don't work with name/value pairs, but only with values. As with Kernel Calls, there is no formal way to identify, which values are input and which are output.

```
REPORT ZSYSTEM_CALL.
DATA: lv_name       TYPE string.
DATA: lv_value      TYPE string.
DATA: m_last_error TYPE i.
DATA: m_c_msg       TYPE %_c_pointer.

lv_name = 'chevron_V'.

SYSTEM-CALL ICT
  DID
    11
  PARAMETERS
    m_c_msg
    lv_name
    lv_value
    m_last_error.
```

System Calls, too, are deprecated. The SAP documentation says that as of Release 6.20, Kernel Methods should be used instead of SYSTEM-CALLs. There is no (comprehensive) documentation which System Calls exist, what their functionality is and which parameters can actually be interchanged per individual System Call.

3. Kernel Methods

Kernel Methods are ABAP methods that use the BY KERNEL MODULE *mod* addition. This addition indicates that instead of processing the ABAP method the usual way, a Kernel Call to the specified module is to be executed. Kernel methods must be empty, i.e. they must not contain ABAP code. The Kernel Methods can access the parameters passed to the ABAP method and interchange data with other ABAP programs through them.

```
METHOD convert_abuf BY KERNEL MODULE ab_expconvDbuf.
ENDMETHOD.
```

SAP customers can’t create custom Kernel Methods, since the compiler checks in a look-up table if the given kernel module is registered for the implementation of the given method / class. However, developers can invoke a Kernel Method the same way they invoke a normal method.

Brief summary: Certain ABAP commands can execute special functions in the SAP Kernel and exchange data with them. This means they are directly communicating with C/C++ functions.

Risk & Relevance: HIGH. Custom ABAP coding may (secretly) make use of undocumented Kernel Calls and this way bypass security features of the SAP standard. I will give several examples of dangerous Kernel Calls in a later section.

Kernel communication is shown in Figure 5.

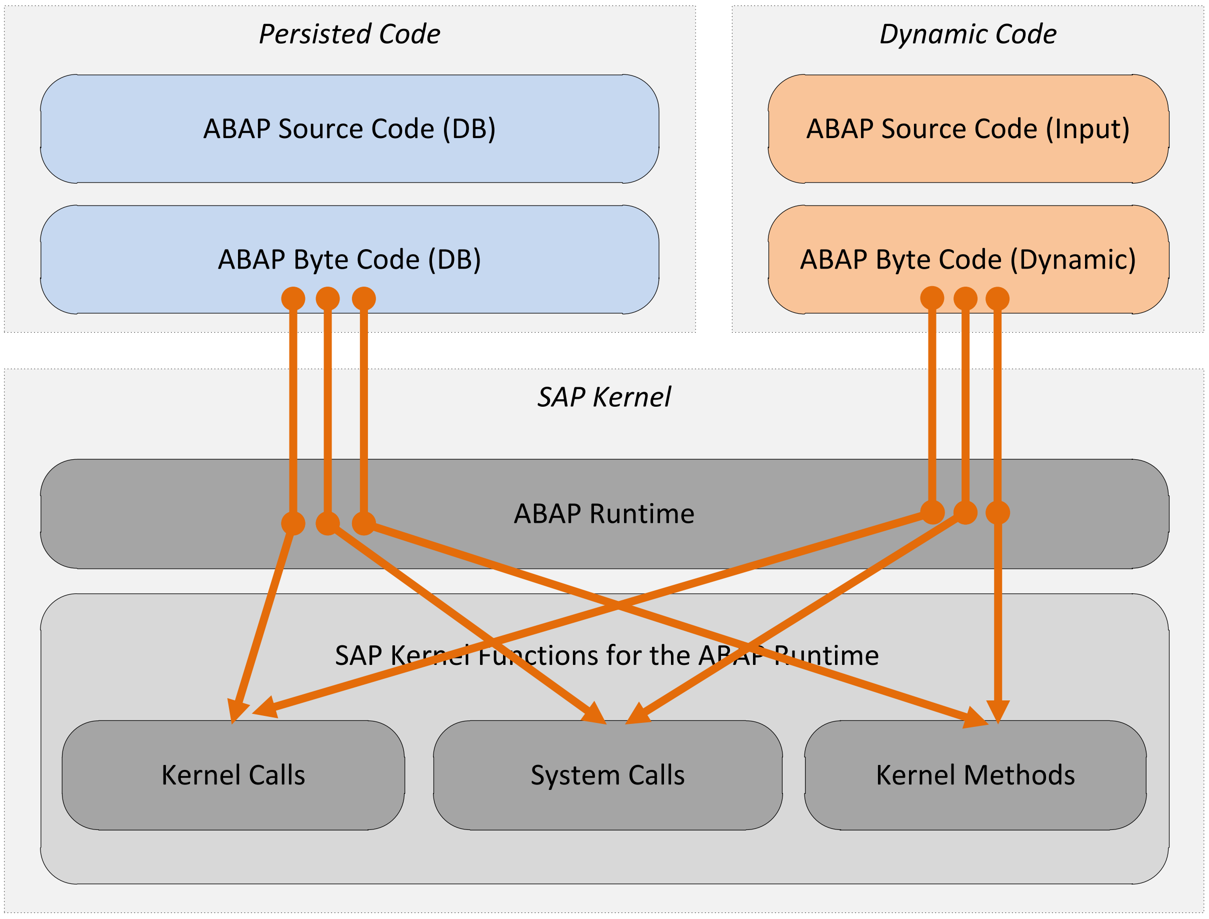

Figure 5: Interaction with the SAP Kernel

Kernel Hooks

In the previous section we have seen that the processing of the ABAP byte is not entirely performed by the ABAP Runtime: certain ABAP commands can actively trigger special functions in the *SAP Kernel*. This section shows that the ABAP Runtime in some instances calls special functions in the *ABAP Byte Code* when parsing certain ABAP commands.

For example, when an OPEN DATASET command is processed (which is used to open a file on the SAP server), the ABAP Runtime at some point during processing calls the ABAP subroutine SYSTEM_HOOK_OPEN_DATASET (function group SYST). This form is designed to check (additional) authorizations for file access. Specifically, it checks for proper S_PATH authorization.

Another example for such a Kernel Hook is form UTC_TIMEZONE_OFFSET (program SAPCNVTS), which is called when the ABAP commands CONVERT TIME-STAMP, CONVERT INTO TIME-STAMP and WRITE AS TIME-STAMP are processed.

The concept of a Kernel Hook is allowing ABAP developers (from SAP) to modify / enhance the behavior of the ABAP Runtime (written in C/C++) without the need to change the SAP Kernel.

Kernel Hooks exchange data with the ABAP Runtime through the Kernel Calls ‘AB_GET_C_PARMS’ (which reads data from the Kernel) and ‘AB_SET_C_PARMS’ (which writes data back to the Kernel). The following program illustrates the principle of this data exchange.

```
FORM SYSTEM_HOOK_ZFT.

FIELD-SYMBOLS:  <P1> type any,
                <P2> type any,
                <P3> type any,
                <P4> type any,
                <R1> TYPE I,
                <R2> LIKE SY-SUBRC.

* First, pop the parameters provided by the Kernel from the stack

CALL 'AB_GET_C_PARMS' ID 'P1' field <P1>
                      ID 'P2' field <P2>
                      ID 'P3' field <P3>
                      ID 'P4' field <P4>.

* Next, call a function to process the data

CALL FUNCTION 'ZPROCESS_THIS_STUFF'
  EXPORTING date  = <P1>
            time  = <P2>
            hour  = <P3>
            mint  = <P4>
  IMPORTING res   = <R1>
            subrc = <R2>
  EXCEPTIONS NO_PERMISSION.

* Finally, push the results to the Kernel stack and exit the hook

CALL 'AB_SET_C_PARMS' ID 'P1' FIELD <R1>
                      ID 'P2' FIELD <R2>.

ENDFORM.
```

Risk & Relevance: HIGH. If a Kernel Hook is modified in the SAP Basis code, it is possible to install code that is called every time a given ABAP command is executed. This would e.g. allow hooking into the execution of OPEN DATASET. This way, all file access in the SAP system could be monitored or even blocked. Whoever experiments with this: don’t call the ABAP command you’re hooked in…

The concept of Kernel Hooks is visualized in Figure 6.

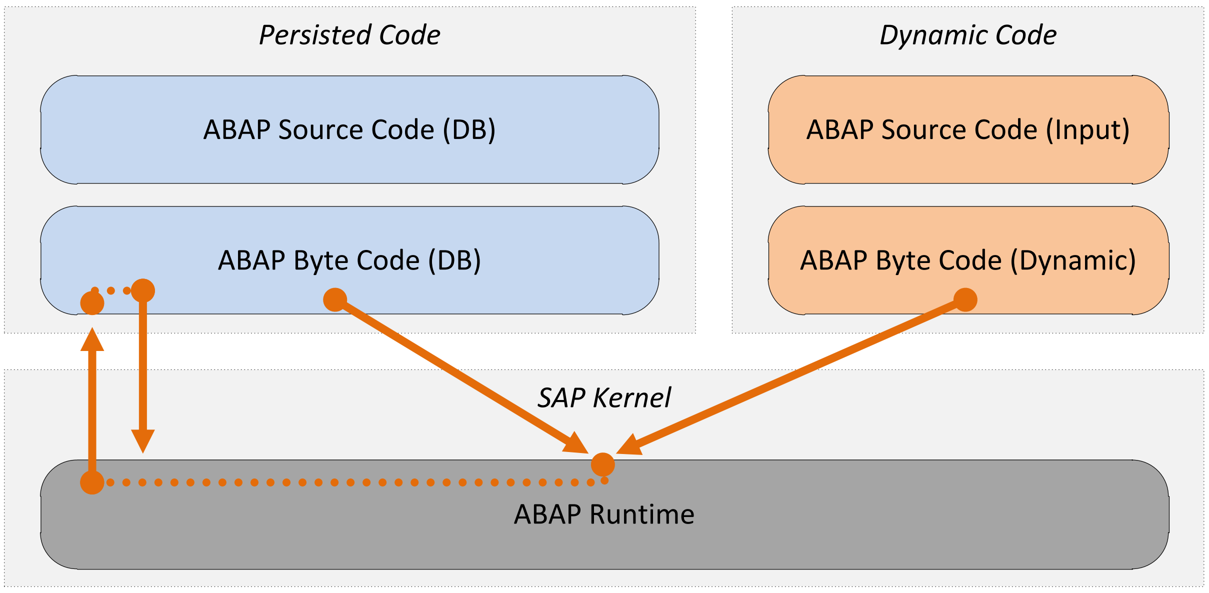

Figure 6: Kernel Hooks calling back into the Byte Code

<u>Summary of ABAP Runtime behavior</u>

1. Not all ABAP statements are static.
2. Dynamic ABAP statements can be (partially) manipulated by malicious input.
3. ABAP programs can be dynamically created, stored and executed based on user input.
4. Custom ABAP code can (secretly) invoke undocumented (and risky) Kernel Functions.
5. The ABAP Runtime provides “hooks” for certain ABAP commands that invoke ABAP Code.

Risky ABAP Kernel Calls

"It is the lure of the labyrinth that draws us to our chosen field."

As mentioned before, there are more than 370 different Kernel Calls in the SAP standard. Most of them have no (comprehensive) documentation and their effects are obscure. During my analysis in the past years, I came across several Kernel Calls that are risky, since they have the potential to bypass security mechanisms in the SAP standard. Some of the risky calls are described in the following, some others are still in "quarantine" due to responsible disclosure.

1. CALL 'SYSTEM'

SAP has provided a standard method to execute OS commands from ABAP, if there is a need. The transactions SM49 and SM69 allow administrators to configure a white list of allowed OS commands that can be executed on an SAP system. Additionally the privilege to execute commands of this defined white list is restricted by authorization object S_LOG_COM. Only users with S_LOG_COM privileges can execute logical OS commands defined with SM49/SM69, which will be subsequently resolved to the corresponding real OS commands.

However, the Kernel Call 'SYSTEM' undermines this secure standard mechanism. It allows execution of arbitrary OS commands on the SAP application server. No matter if these are defined in the white list of SM49/SM69 or not. CALL 'SYSTEM' is independent of S_LOG_COM and therefore an extremely dangerous command and should not be used / allowed in (custom) ABAP code.

```
REPORT Z_CALL_SYSTEM.

TYPES lt_line(255) TYPE c.

DATA lv_cmd(42) TYPE c.
DATA lt_result  TYPE STANDARD TABLE OF lt_line WITH HEADER LINE.

* Simple example that reads a directory listing.

lv_cmd = 'ls -a'.

CALL 'SYSTEM' ID 'COMMAND' FIELD lv_cmd
              ID 'TAB' FIELD lt_result-*sys*.

LOOP AT lt_result.
  WRITE : / lt_result.
ENDLOOP.
```

On the upside, execution of this particular Kernel Call is one of the rare cases, where an implicit authority check (S_C_FUNCT) is performed by the Kernel. Users that are not explicitly assigned this authorization can't execute CALL 'SYSTEM'. As an additional measure, it is recommended to disable this Kernel Call entirely by setting profile parameter 'rdisp/call_system' to '0' (zero). However, even SAP standard coding uses this Kernel Call in many programs and may fail, if it is disabled…

Risk & Relevance: HIGH. Usage of this Kernel Call may bypass the OS command white list configured in SM49/69. A successful exploit may result in complete compromise of the operating system of the SAP application server.

2. CALL ‘XXPASS’ and CALL ‘XXPASSNET’

These Kernel Calls are used to compute the password hash for SAP logon, which is stored in table USR02. This Kernel Call comes in handy, if a rainbow table for the password hash is to be built. However, if a custom program tries to use these Kernel Calls, the session is immediately ended and the current user account is locked. Nice. The ABAP Runtime obviously checks the name(space) of the calling program, since the SAP standard uses this Kernel Call several times:

• Include MS01JF10, Form CHECK_PASS	(XXPASS)
• Include MS01JF10, Form SET_NEW_PASS	(XXPASS)
• Include MS01CD10_DEL_USR04, Form DEL_USR04	(XXPASS)
• Function module SUSR_USER_PASSWORD_PUT	(XXPASS)
• Function group SUU5, Include LSUU5I01	(XXPASS)
• Function group ALEW, Form P_ENCODE	(XXPASSNET)

This account lockout can be prevented, if the name of the program that performs the CALL ‘XXPASS’ is outside of the usual ‘Y’ and ‘Z’ namespace for customers. Such a namespace (e.g. /VFORGE/) needs to be registered with SAP. Note that, even if the lockout is avoided this way, CALL ‘XXPASS’ does not return meaningful output (hash is zero-filled).

This mechanism is interesting, as the Kernel prevents usage of certain Kernel Calls by checking the name(space) of the calling program/module rather than performing an authority check. This indicates that some Kernel Calls are so critical, no one (outside) SAP may use them.

```
REPORT Z_LOCKME.

  DATA lv_hash TYPE usr02-bcode.
  DATA lv_pass TYPE usr02-passcode.
  DATA lv_vers TYPE usr02-codvn.

* Kill my session and lock my account

  CALL 'XXPASS'
    ID 'CODE'     FIELD 'akagi'
    ID 'NAME'     FIELD 'castle'
    ID 'CODX'     FIELD lv_hash
    ID 'PASSCODE' FIELD lv_pass
    ID 'VERS'     FIELD lv_vers.
```

Risk & Relevance: MEDIUM. Usage of this Kernel Call will lockout the current user account, if called from a custom ABAP program in Z/Y namespace.

3. CALL 'INTERNET_USER_LOGON'

In several SAP Web scenarios, users are initially logged in under an alias user account. This is necessary, if those users need to view business data without logging in. E.g. SAP eRecruitment makes use of this feature. However, at some point in time the user may decide to login. Then the SAP system needs to perform a "user switch", i.e. login the user again, but this time under his/her own account. This "user switch" is done by Kernel Call 'INTERNET_USER_LOGON'. This Kernel Call attempts to logon the given user name with the given password. It can also create a SAP logon ticket, if requested. Of course, incorrect user names and passwords are handled the same way as during standard login. And the account is locked, in case the number of failed login attempts exceeds the maximum configured.

```
REPORT Z_LOGON.

* Simple example, using some of the parameters.

CALL 'INTERNET_USER_LOGON' ID 'UNAME'    FIELD lv_userid
                           ID 'PASSW'    FIELD lv_password
                           ID 'TICKET'   FIELD lv_ticket
                           ID 'PASSFLAG' FIELD lv_pwdstate.
```

However, since this Kernel Call can be used by ABAP programs, it may be used to launch brute force password guessing attempts. Since the number of failed logins as well as the account lockout flag are stored in table USR02, a malicious (custom) ABAP program may use CALL 'INTERNET_USER_LOGIN ' and in each failed instance reset the number of failed login and/or unlock the user account.

This Kernel Call is executed by the following SAP standard modules:

- Function module SUSR_INTERNET_USERSWITCH
- Function module SUSR_CHECK_LOGON_DATA
- Function module WSS_INTERNET_USERSWITCH
- Class CL_ME_SYNC_GET_DATE_VALUE, Method IF_HTTP_EXTENSION~HANDLE_REQUEST
- Class CL_HTTP_EXT_ECHO, Method IF_HTTP_EXTENSION~HANDLE_REQUEST
- Class CL_USER_POC, Method LOGIN
- Class CL_WSSE_CONTEXT, Method AUTHENTICATE

Note that 'INTERNET_USER_LOGON' can also be called in 'TESTMODE'. In this case, the user is not logged in / switched, but only the credentials are checked. Function module 'SUSR_CHECK_LOGON_DATA' makes use of this testmode functionality.

Risk & Relevance: LOW. This Kernel Call can be misused to launch a brute force password cracking attempt from a custom ABAP program. However, using an external brute force tool (e.g. John The Ripper) on an extract of table USR02 may be more efficient. But if SAP changes their password algorithm again (as they have done multiple times in the past), this Kernel Call might come in handy.

4. CALL ‘C_GET_TABLE’

Although the standard way to access the SAP database is Open SQL, SAP provides alternative ways to read data from the database. The command CALL ‘C_GET_TABLE’ reads the contents of the given database table. Since it is possible to provide the complete query key, all data from the given table can be read, i.e. the data of all SAP clients.

There are no implicit authorization restrictions regarding this Kernel Call.

```
REPORT Z_GETTAB.

* Simple example for a table read

CALL 'C_GET_TABLE' ID 'TABLNAME'  FIELD lv_tab
                   ID 'INTTAB'    FIELD lt_tab-*sys*
                   ID 'GENKEY'    FIELD lv_key
                   ID 'GENKEY_LN' FIELD lv_keylen
                   ID 'BYPASS'    FIELD lv_bypass.
```

This Kernel Call is executed by the following SAP standard modules:

- Function module RFC_GET_TABLE_ENTRIES
- Function module DB_SETGET
- Function module DB_SELECT_GENERIC_TABLE
- Function group SCAT, Form CHECK_C

Risk & Relevance: HIGH. This Kernel Call is equivalent to the functionality of transaction SE16, but without authority checks. If it is used (improperly) in custom code, an unauthorized user may gain read access to arbitrary data in the SAP database. This data access is cross-client.

5. CALL ‘C_MOD_TABLE’

This is another command that bypasses Open SQL database access. In this case, a database table can be modified with the values provided in an internal table. There are three options to modify the table: (I)nsert, (U)pdate or (D)elete rows.

There are no implicit authorization restrictions regarding this Kernel Call.

```
REPORT Z_MODTAB.

* Simple example for a database (U)pdate.

CALL 'C_MOD_TABLE' ID 'TABLNAME' FIELD lv_tab
                   ID 'INTTAB'   FIELD lt_tab-*sys*
                   ID 'FCODE'    FIELD 'U'
                   ID 'DBCNT'    FIELD lv_dbcnt
                   ID 'SQLCODE'  FIELD lv_sqlcode.
```

This Kernel Call is executed by the following SAP standard modules:

- Function module DB_UPDATE_TABLE
- Function module DB_INSERT_TABLE
- Function module DB_DELETE_TABLE

Risk & Relevance: HIGH. This Kernel Call can be used to directly create/modify/delete data in SAP tables. The data access is cross-client. If used improperly in custom ABAP code, unauthorized users are able to modify arbitrary data in the SAP database. Also, no change documents are written, in case of this low-level access.

6. CALL ‘C_DB_EXECUTE’

While the two previous Kernel Calls can perform read / write access to any given table, CALL ‘C_DB_EXECUTE’ can actually execute *arbitrary* SQL statements on the SAP database (with the exception of SELECTs). This functionality is very dangerous, since it allows executing commands far beyond the Open SQL capabilities in the SAP standard, like e.g. a DROP TABLE.

This Kernel Call is executed by the following SAP standard modules (excerpt):

- Function module DB_EXECUTE_SQL
- Function module RSDU_EXEC_SQL_INF

There are no implicit authorization restrictions regarding this Kernel Call.

```
REPORT Z_SQL_EXECUTE.

* Simple example for a generic SQL command.

CALL 'C_DB_EXECUTE' ID 'STATLEN' FIELD lv_len
                    ID 'STATTXT' FIELD lv_stmt
                    ID 'SQLERR'  FIELD lv_sqlerr.
```

Risk & Relevance: VERY HIGH. This Kernel Call can be used to execute *arbitrary* database commands for the sole exception of SELECT statements. If it is used (improperly), unauthorized users may gain *full control* over the SAP database, even to areas that Open SQL commands can’t touch. This Kernel Call MUST NOT be used in custom code.

7. CALL 'C_DB_FUNCTION'

This Kernel Call not only allows executing *arbitrary* SQL statements, it also gives access to stored procedures of the database. This functionality is very dangerous, since it allows executing commands far beyond the Open SQL capabilities in the SAP standard, like e.g. a DROP TABLE or executing a dangerous stored procedure. This Kernel Call is frequently used in the SAP standard and is a core component of SAP's ABAP Database Connectivity (ADBC) [see package SDB_ADBC]. This function module is rather complex and my research is ongoing.

There are no implicit authorization restrictions regarding this Kernel Call.

```
REPORT Z_DB_EXECUTE.

* Simple example for a generic stored procedure call.

CALL 'C_DB_FUNCTION' ID 'FUNCTION' FIELD 'DB_SQL'
                     ID 'FCODE'    FIELD 'EP'
                     ID 'PROCNAME' FIELD lv_procedure
                     ID 'CONNAME'  FIELD lv_con
                     ID 'CONDA'    FIELD lv_conda
                     ID 'PARAMS'   FIELD lt_params
                     ID 'ROWCNT'   FIELD lv_rows_processed
                     ID 'SQLCODE'  FIELD lv_sql_code
                     ID 'SQLMSG'   FIELD lv_sql_msg.

* Simple example for a generic SQL command.

CALL 'C_DB_FUNCTION' ID 'FUNCTION'    FIELD 'DB_SQL'
                     ID 'FCODE'       FIELD 'PO'
                     ID 'STMT_STR'    FIELD lv_statement
                     ID 'CONNAME'     FIELD lv_con
                     ID 'CONDA'       FIELD lv_conda
                     ID 'HOLD_CURSOR' FIELD lv_cursor
                     ID 'INVALS'      FIELD lz_params
                     ID 'CURSOR'      FIELD c
                     ID 'SQLCODE'     FIELD lv_sql_code
                     ID 'SQLMSG'      FIELD lv_sql_msg.
```

Risk & Relevance: VERY HIGH. This Kernel Call can be used to execute *arbitrary* SQL commands, database commands and stored procedures. If it is used (improperly), unauthorized users may gain *full control* over the SAP database, even to areas that Open SQL commands can't touch. This Kernel Call MUST NOT be used in custom code.

Summary of ABAP Kernel Calls

1. There is no (comprehensive) documentation on Kernel Calls.
2. In the jungle of > 370 Kernel Calls, there is an unknown number of critical functionality.
3. There are several risky Kernel Calls that can bypass security mechanisms of the SAP standard.

Buffer Overflows in ABAP Kernel Calls

"Maybe they don't show up on infra-red at all."

All input processed by ABAP programs is initially handled by the ABAP Runtime. Since the ABAP Runtime is a mature and highly tested environment, the risks of a buffer overflow are near zero. The ABAP Runtime is the first line of defense and a shield for the lower levels of the SAP Kernel from input. This is visualized in Figure 7.

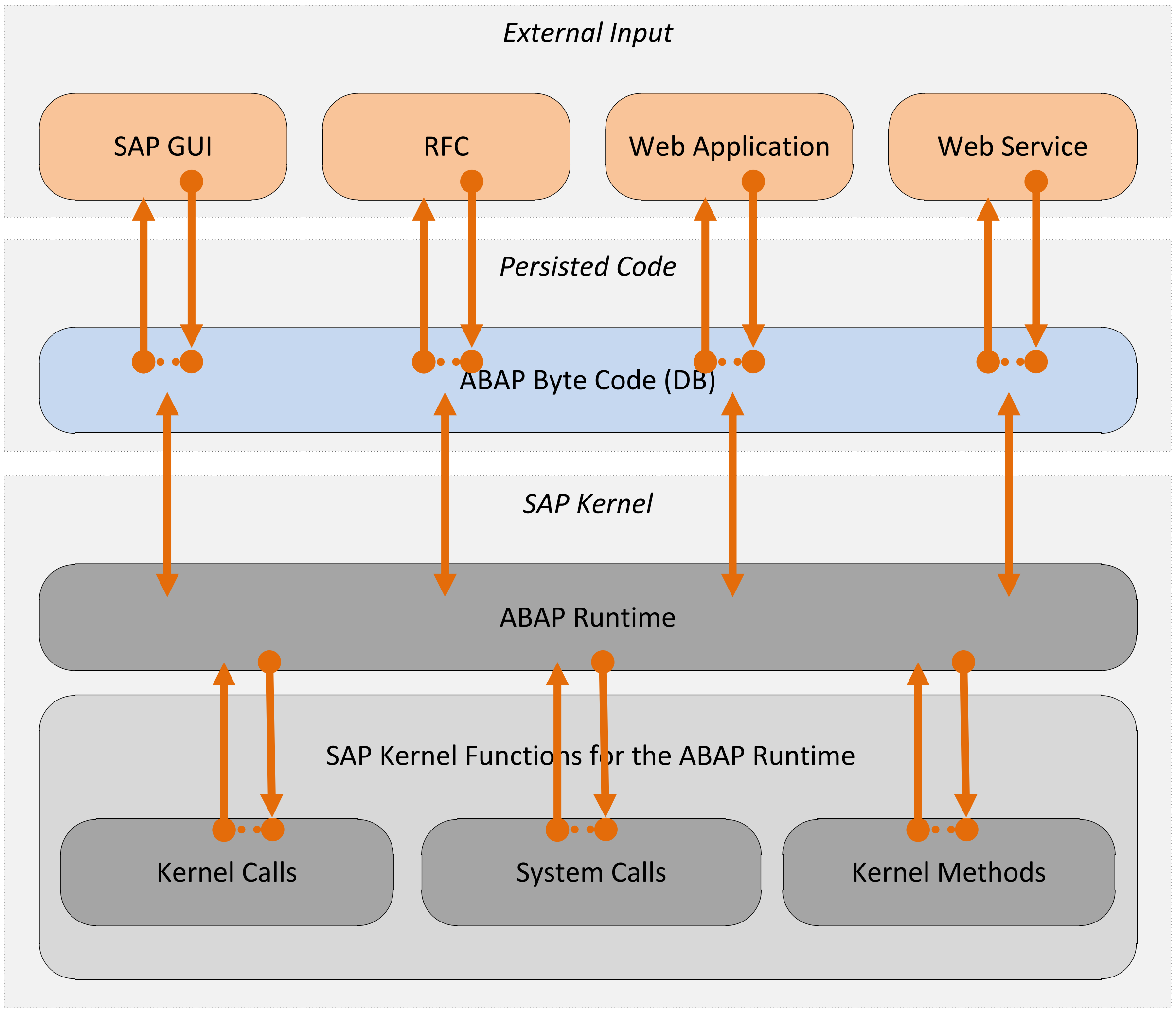

Figure 7: The ABAP Runtime shields Kernel Functions from direct input

In contrast to the ABAP Runtime, Kernel functions have very limited exposure to attack, since they are used comparably rarely. And there usually is no direct route from the outside to the parameters of a SAP Kernel Function. Unless of course an ABAP program directly passes on its input to a Kernel Function. (Shown in Figure 8)

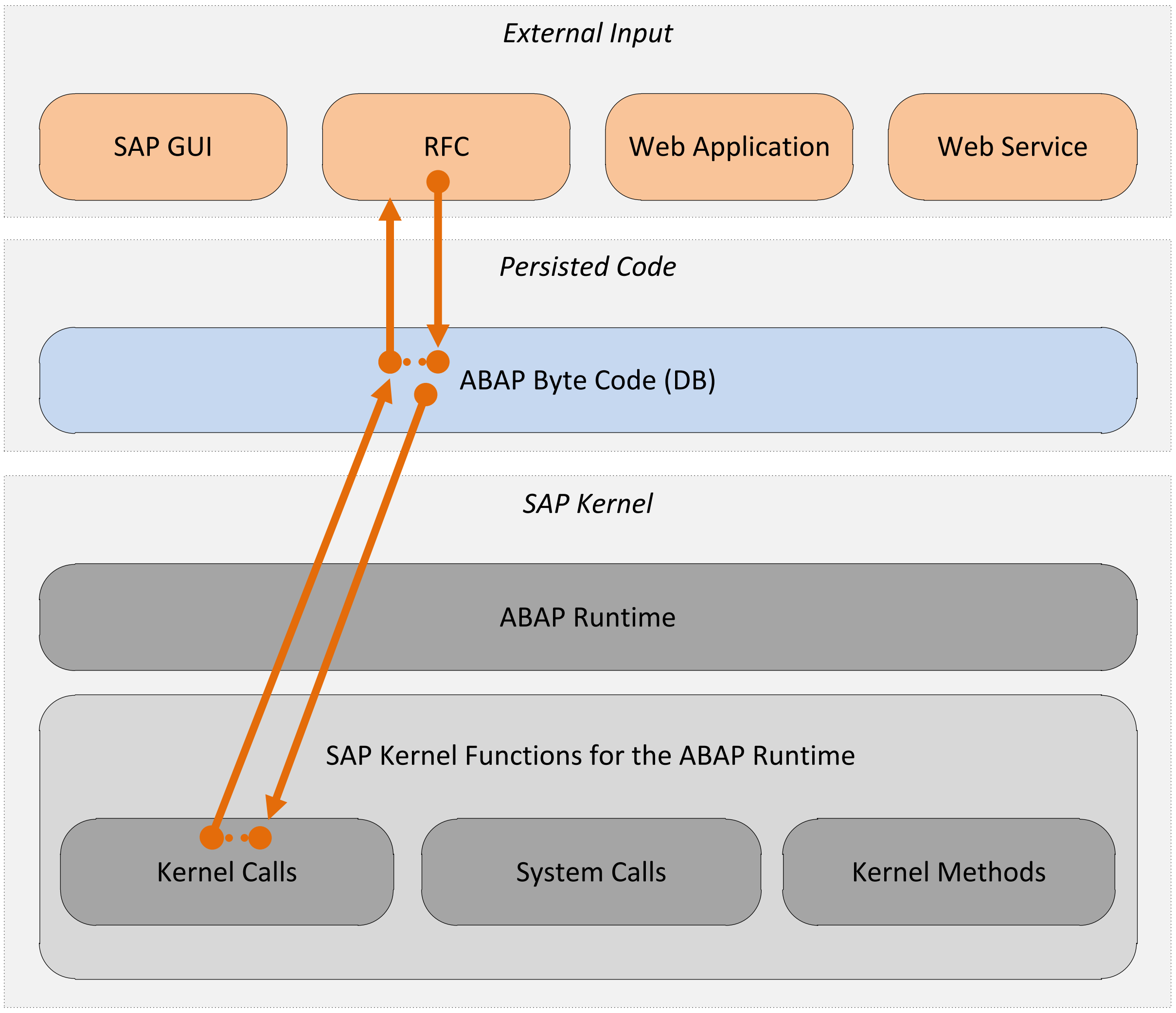

Figure 8: RFC-enabled ABAP Code directly passes its input directly to a Kernel Call.

If external input is directly passed on to a vulnerable SAP Kernel Function (using ABAP as a tunnel), then a buffer overflow may occur. As a precondition, the ABAP variable that holds the input must be large enough to trigger the overflow.

In the past Virtual Forge researchers have discovered multiple buffer overflows in SAP Kernel Calls as well as a buffer overflow in a SAP System Call. Some of those are resolved [see SAP security notes (3) and (4) in references section], some are not (yet).

Most of the buffer overflows in vulnerable Kernel Calls we discovered were not exploitable in the SAP standard and only pose a risk when used in custom ABAP code. A vulnerably Kernel call becomes exploitable only if external input is fed to a critical parameter and if the variable transporting the external input is big enough for an attack. Therefore, any usage of Kernel Calls / System Calls / Kernel Methods in custom code can expose dormant buffer overflows to external input. Kernel Methods, however, are to some degree better protected than Kernel Calls and System Calls. This is because the Kernel Method can specify the parameter type explicitly and thereby restrict the length of their input values. Even vulnerable Kernel Methods could be protected this way.

We discovered buffer overflow vulnerabilities in the following Kernel Calls
(vulnerable parameters highlighted in red):

```
* This Kernel Call reads profile parameters (customized values)
CALL 'C_SAPGPARAM' ID 'NAME' FIELD lv_buffy
                   ID 'VALUE' FIELD lv_dummy.

* This Kernel Call reads profile parameters (default values)
CALL 'C_SAPGDEFPARAM' ID 'NAME' FIELD lv_buffy
                      ID 'VALUE' FIELD lv_dummy.

* Creates an OS specific
CALL 'BUILD_DS_SPEC' ID 'FILENAME' FIELD lv_buffy
                     ID 'PATH' FIELD lv_buffy
                     ID 'OPSYS' FIELD lv_buffy
                     ID 'RESULT' FIELD lv_dummy.
```

The affected System Call is:

```
SYSTEM-CALL ICT
  DID
    29
  PARAMETERS
    lv_buffy
    lv_buffy
    lv_dummy
    lv_dummy
    lv_dummy
    lv_dummy
    lv_dummy
    lv_dummy.
```

Remote-exploitable buffer overflow vulnerability in a SAP Kernel Call.

Only in one instance external input of significant length was fed to a vulnerable Kern Call. A remote exploitable buffer overflow vulnerability existed for CALL ‘C_SAPGPARAM’, which is (among other locations) exposed by the RFC-enabled function module **RSPO_R_SAPGPARAM**. This function module exposes parameter NAME (of the Kernel Call) to external input of 255 characters length. This was sufficient to trigger the overflow. (SAP Security Note 1487330 addresses the three vulnerable Kernel Calls above.)

Summary

"Understanding is a three-edged sword."

Even the most secure SAP system can easily be compromised by insecure ABAP programs:

- Dynamic ABAP commands can introduce critical side effects like Open SQL Injection.
  (Results in unauthorized access to business data)
- Dynamic ABAP code generation gives users complete control over the SAP server.
  (Results in complete compromise of the SAP server, including easy access to all data)
- Missing/Improper authorization checks in ABAP code can give unauthorized users to restricted business functionality
  (Results in privilege escalation)
- Malicious users can elevate their privileges on a productive SAP system by coding a backdoor into an ABAP program.
  (Results in fraud)
- Kernel Communication allows ABAP programs to bypass critical security mechanisms of the ABAP language / the SAP standard.
  (Results in privilege escalation)
- Using Kernel Calls and System Calls in ABAP programs can expose dormant buffer overflows to external input.
  (Results in complete compromise of the SAP server)

All of the above security problems result in a fault in *IT general controls*. Consequently, all compliance standards that consider IT general controls as a basis for audits will be violated by insecure ABAP code. This includes SOX, EuroSOX, FDA and BASEL II.

Important advice to the best-run companies: Thoroughly check all ABAP coding before running it on your productive servers!

Disclaimer

“Mul-ti-pass.”

References and further reading

"The answers are there. You just have to know where to look."

Material available to the general public

(1) "Sichere ABAP Programmierung" (Secure ABAP Programming) by Wiegenstein, Schumacher, Schinzel, Weidemann @ SAP Press 2009 (available only in German)
(2) BIZEC - The Business Security Initiative @ http://www.bizec.org

Material available only to SAP customers (via SAP Service Marketplace)

(3) SAP Security Note 1493516 - Correcting buffer overflow in ABAP system call
(4) SAP Security Note 1487330 - Potential remote code execution in SAP Kernel

About Virtual Forge

"We harden your software."

Virtual Forge GmbH is an independent security product company based in Heidelberg, Germany. Our employees are leading experts in the area of SAP application security. Our unique ABAP security knowledge has been captured into CodeProfiler, the first static code analysis tool with data- and control-flow capabilities for ABAP security and compliance testing. CodeProfiler and related products of our ABAP security suite enable companies to develop business applications that meet state of the art security and compliance standards.

Worldwide, companies running SAP have improved their ABAP development lifecycle with Virtual Forge's ABAP security suite. Our solutions are applied in industries such as defense, pharmaceuticals, banking, oil & gas, automotive, engineering, health care, agriculture, and insurance. Virtual Forge also cooperates with renowned ISVs.

For further information about our services and solutions, you are welcome to visit our website at http://www.virtualforge.com.