

Segurança ABAP – Parte I – Código dinâmico

02/07/2012 14:00

Fala zombizada! Aqui quem fala é o seu camarada [@ABAPdepressao](#), vindo diretamente do mundinho do Twitter onde compartilho das tristezas diárias de programar em alemão com vocês. O Maurício me convidou pra escrever aqui no ABAP Zombie, e eu tentei escolher um tema bacana pra não estragar logo de cara o blog com o meu humor depressivo (ou deprimente). Então vou tentar falar de segurança dentro do ABAP.



"Meu nome é AUTHORITY-CHECK, e o seu SY-SUBRC é DOZE"

Segurança da informação é um assunto que me interessa bastante. Essa é uma área crítica em qualquer sistema, e dentro do SAP não é diferente. Muitos elementos já estão ali para garantir a segurança, como o modelo de autorização, a criptografia e as políticas de senha, o transporte e versionamento de código, a separação de sistemas no *landscape* (DEV>QAS>PRD), *Single Sign-On*, *patches* de segurança, etc. Todas essas coisas, quando bem administradas, tornam o SAP um sistema seguro.

Quando eu comecei a trabalhar com SAP, ainda estagiário de Basis, eu não cheguei a mexer com tudo isso, mas só com a parte de administração de usuários e perfis. Sempre tinha muito trabalho pra fazer, porque o cliente tinha políticas bem definidas para as contas de usuários e segregação de funções. Só que depois que eu passei pra área de ABAP, eu percebi que não havia o mesmo cuidado dentro do código. A equipe de Basis era responsável por garantir acesso às transações e aos programas desenvolvidos, mas uma vez dentro dos mesmos o máximo que acontecia era um AUTHORITY-CHECK aqui e outro ali.

Isso me espantou de início, mas depois eu entendi que o *standard* é quem implementa a maior parte da segurança dentro do SAP, através dos elementos que eu citei anteriormente. É claro que isso não tira a responsabilidade do ABAPer de usar AUTHORITY-CHECKs nos lugares necessários, mas acaba tirando um grande peso das costas do desenvolvimento, diferentemente do que acontece em outras linguagens.

Mas quando falamos de **vulnerabilidades**, sim, o ABAP é parecido com outras linguagens. Fuçando na net eu encontrei um [white paper](#) que fala sobre o tema. É um estudo de uma empresa de segurança de *software* alemã chamada VirtualForge sobre os riscos do ABAP, mais especificamente sobre comunicação com o *kernel* do SAP e

buffer overflows. São 27 páginas com alguns exemplos, então é um documento razoavelmente extenso, porém interessante.

Antes de me prolongar sobre o tema, é importante ressaltar o seguinte: **todas** as vulnerabilidades que estão expostas nesse documento podem ser (pra usar o jargão de segurança :D) mitigadas dentro dos desenvolvimentos através de: 1- controle do sistema bem exercido pela equipe de Basis e 2- *peer review*, a famigerada revisão por pares.

O documento classifica as vulnerabilidades pelo risco (baixo / médio / alto / muito alto). Não vou falar de todas elas agora – fica pras próximas partes – mas vou começar com uma que acho bastante complicada: **código dinâmico**.

Por “código dinâmico” entenda-se código que não é totalmente compilado, porque possui partes variáveis que só são determinadas em tempo de execução. Os casos mais simples de código dinâmico em ABAP são certos comandos que aceitam referências como parâmetros. Vejamos dois exemplos:

1- Usando ASSIGN:

```
DATA: lv_segredo TYPE string VALUE 'Segredo',
      lv_aberto  TYPE string VALUE 'Aberto'.

PARAMETERS p_ref TYPE string DEFAULT lv_aberto.

FIELD-SYMBOLS <fs_ref> TYPE ANY.

ASSIGN (p_ref) TO <fs_ref>.
```

2- Usando OpenSQL:

```
PARAMETERS p_where TYPE string DEFAULT bname = sy-uname.

DELETE FROM usr01 WHERE (p_where).
```

Ao passar um parâmetro entre parênteses, não é a variável declarada diretamente que está sendo usada, mas sim o valor dela. Isso pode abrir um precedente para que esses valores sejam alterados em tempo de execução, permitindo assim que o programa leia dados que não deveria (caso do ASSIGN, se mudarmos o valor da P_REF), ou execute comandos usando dados diferentes dos que a lógica da aplicação pretendia usar (caso do DELETE, mudando o valor da P_WHERE).

Isso já seria dor de cabeça suficiente para um Basis, principalmente quando falamos de interação com o banco de dados. Mas como se não bastasse, existe também um comando nem tão conhecido que pode gerar código **totalmente** dinâmico: o **GENERATE SUBROUTINE POOL**. Exemplo:

```
REPORT z_codigo_dinamico.

DATA: t_pool TYPE TABLE OF string,
      v_prog TYPE string.


APPEND: 'PROGRAM pool.' TO t_pool,
        'FORM mensagem.' TO t_pool,
        '  WRITE / ''Código dinâmico chamado''.' TO t_pool,
        'ENDFORM.' TO t_pool.

GENERATE SUBROUTINE POOL t_pool NAME v_prog.

WRITE: 'Nome do programa gerado: ', v_prog.

PERFORM ('MENSAGEM') IN PROGRAM (v_prog) IF FOUND.
```

Exemplo de saída do programa:

Nome **do** programa gerado: _T00E10

Código dinâmico chamado

Impressionou? Não? Então vamos analisar duas coisas:

1. O código criado com GENERATE SUBROUTINE POOL é transformado em *byte code* em tempo de execução, **sem nunca ser gravado no banco de dados** como acontece com um programa comum (feito através da SE38, SE24, SE80, etc);
2. O nome do *subroutine pool* é gerado dinamicamente e começa com "%_", portanto **está fora dos namespaces do cliente** (Z*, Y*, /NAMESPACE_DO_CLIENTE/*). Dessa forma, investigar o funcionamento do código dinâmico através de *traces* fica muito mais difícil.

Não precisa juntar nem 2 + 2 pra entender que se a geração de código dinâmico estiver sujeita a *input* externo (seja por tela de seleção, leitura de arquivo, interface PI, ABAP Proxy, etc.) o programa pode fazer **qualquer coisa**. Isso é um risco **MUITO GRANDE**.



Isso também é um risco muito grande, mas esse sai com Colorcote 2000. NOSSA, NOSTALGIA FORTE

Eu sei que podem existir situações muito complexas em que essas técnicas se fazem necessárias, mas a moral aqui é bem evidente: **evite usar código dinâmico**, e se tiver que usar, **policie muito bem as entradas**. Seu amigo (?) Basis agradece quando rolar aquela auditoria.

Até a próxima, onde vou tentar falar sobre os *kernel calls*.

Comentários

roberto florentino — 29/08/2013 15:23

Matéria muito interessante, despertou a necessidade de termos alguma pessoa cuidando da questão da qualidade dos desenvolvimentos em abap, com foco na segurança e performance

Jahniffer Santos — 06/06/2013 13:45

kkkk... (...Isso é um risco MUITO GRANDE...)... Muito legal. Também gostei muito do artigo. Uma pitadinha de Basis no ABAP para ajudar a segurança. Viva o compartilhamento do conhecimento!!!

Diego Henrique Gomes — 02/07/2012 23:29

[illegible]

Mauricio Cruz — 02/07/2012 15:10

Só pra constar, vale a pena ler o White Paper que o mano depressivo citou. Eu já trabalhei com uma galera que manjava dessa parte de segurança em ABAP, e é bem interessante analisar sua aplicação desse outro ponto de vista 😊

E que venha a parte 2!

Abraços!