




Análise Estática e Protecção Dinâmica

Parte III: Técnicas de Protecção

Segurança de Software
2019
Nuno Santos



Onde estamos

- ▶ **Parte I: Enquadramento e protecção (2 aulas)**
 - ▶ Conceitos de segurança de software, mecanismos básicos de segurança
- ▶ **Parte II: Vulnerabilidades (3 aulas)**
 - ▶ Buffer overflows, corridas e validação de entradas, vulnerabilidades na web e em bases de dados
- ▶ **Parte III: Técnicas de protecção (3 aulas)**
 - ▶ Auditoria e teste de software, análise estática de código, protecção dinâmica, desenvolvimento de software seguro

▶ 2 SS - Nuno Santos 2019



Plano para esta aula

- ▶ Técnicas de análise estática
- ▶ Técnicas de protecção dinâmica



Análise estática



Motivação para análise estática

- ▶ “So why do developers keep making the same mistakes? (...) Instead of relying on programmers’ memories, we should strive to produce **tools that codify what is known about common security vulnerabilities and integrate it directly into the development process.**”
 - ▶ David Evans e David Larochelle, Improving Security Using Extensible Lightweight Static Analysis



Análise estática de código fonte

- ▶ **Objectivo: encontrar vulnerabilidades no código fonte das aplicações automaticamente**
 - ▶ Semelhante a um erro de compilador, mas para verificar vulnerabilidades de segurança
 - ▶ Semelhante a inspeção manual do código, mas automático
- ▶ **“Estático” porque o código não é executado**
 - ▶ Algumas ferramentas conseguem analisar código binário ou intermédio (ex. Java bytecode)
 - ▶ Mas analisar código fonte é mais simples, logo, mais comum



Procurar por strings perigosas

- ▶ Ferramentas simples como o `grep` conseguem fazer uma forma muito básica de análise; ex.:

- ▶ `grep gets *.c` `grep strcpy *.c`

- ▶ Limitações:

- ▶ O utilizador tem que saber que funções são perigosas
 - ▶ O utilizador tem que fazer todos os “greps”
 - ▶ Não distingue entre invocações reais de funções perigosas (ex. `gets`) e instâncias dessas strings que não são chamadas

- ▶ Exemplo:

```
int main()
{
    // var. strcpy
    int strcpy;
    return 0;
}
```

▶ 7

SS - Nuno Santos

2019



Fuzzers simples

- ▶ Procuram chamadas sistema / bibliotecas perigosas

- ▶ Ex. `gets`, `strcpy` or `sprintf`

- ▶ Atribuem um nível de perigo às vulnerabilidades potenciais encontradas

- ▶ Exemplos de fuzzers: RATS, Flawfinder, ITS4

- ▶ Principais componentes de um fuzzer destes

- ▶ Base de dados de chamadas sistema/biblioteca perigosas
 - ▶ Preprocessador de código (verifica o que vai ser compilado)
 - ▶ Analizador lexical (lê os nomes das funções)

▶ 8

SS - Nuno Santos

2019



Exemplo de utilização do Flawfinder

Flawfinder version 1.24, (C) 2001-2003 David A. Wheeler.

Number of dangerous functions in C/C++ ruleset: 128

```
...
./teste.cc:96: [4] (buffer) sscanf:
  The scanf() family's %s operation, without a limit specification,
  permits buffer overflows. Specify a limit to %s, or use a different input
  function.
./maisteste.cc:97: [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncpy or strlcat (warning, strncpy is easily misused).
./maisteste.cc:101: [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncpy or strlcat (warning, strncpy is easily misused).
...
```

► 9

SS - Nuno Santos

2019



Exemplo de uma base de dados (Flawfinder)

access	Can lead to process/file interaction race conditions (TOCTOU category A)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
acct	Can lead to process/file interaction race conditions (TOCTOU category A)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
au_to_path	Can lead to process/file interaction race conditions (TOCTOU problems)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
basename	Can lead to process/file interaction race conditions (TOCTOU problems)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
bcopy	At risk for buffer overflows.	Make sure that your buffer is really big enough to handle a max len string.	MODERATE_RISK
bind	potential race condition with access, according to cert. Also, bind(s, INADDR_ANY,) followed by setsockopt(s, SOL_SOCKET, SO_REUSEADDR) leads to potential packet stealing vuln	Be careful.	LOW_RISK
drand48	Don't use rand() and friends for security-critical needs.	Use better sources of randomness, like /dev/random (linux) or Yarrow (windows).	RISKY
erand48	Don't use rand() and friends for security-critical needs.	Use better sources of randomness, like /dev/random (linux) or Yarrow (windows).	RISKY

► 10

SS - Nuno Santos

2019



Falsos positivos

- ▶ **As ferramentas básicas de fuzzing encontram vulnerabilidades**
 - ▶ Melhores que o grep: detectam que o nome da função é uma chamada real
- ▶ **Mas geram muitos falsos positivos**
 - ▶ Ex., *strcpy*, *sprintf*, são perigosas mas o seu uso não resulta obrigatoriamente numa vulnerabilidade
 - ▶ Ex: um pequeno programa feito por alguns alunos foi analisado pelo Flawfinder e pelo RATS, gerando cerca de 80 alarmes, todos falsos
 - ▶ Falsos positivos colocam o fardo de obrigar à verificação manual por um humano
- ▶ **Por isso, o que se pretende nesta área é obter dois objectivos:**
 - ▶ Encontrar todas as vulnerabilidades (sem falsos negativos)
 - ▶ Minimizar o número de falsos positivos



Protecção dinâmica



Motivação

- ▶ Em segurança, métodos passivos não são suficientes
 - ▶ Metodologias de teste, auditoria, análise estática
- ▶ Vulnerabilidades ainda persistem, pelo que precisamos também de protecção dinâmica (ou protecção em runtime)
 - ▶ Estas medidas são provavelmente a explicação para grande redução de ataques de buffer overflow e outras vulnerabilidades de memória nos últimos anos

▶ 13

SS - Nuno Santos

2019



Protecção dinâmica

- ▶ A ideia é bloquear os ataques que poderão estar a tentar explorar vulnerabilidades existentes
- ▶ Consideraremos sobretudo ataques de corrupção de memória
 - ▶ Uma das classes mais frequentes: buffer overflows
 - ▶ Boa ideia assumir que não vão desaparecer completamente
 - ▶ Portanto, ter mecanismos que minimizem o seu impacto é importante
 - ▶ O tópico é muito vasto, existem muitas medidas

▶ 14

SS - Nuno Santos

2019

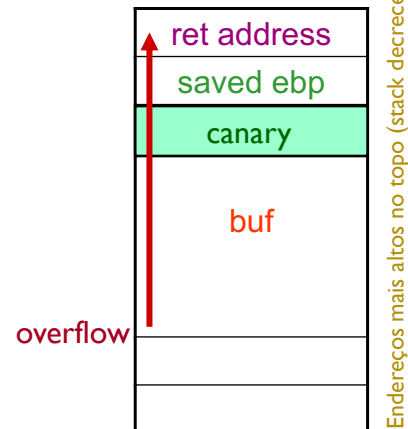


Canários / Stack cookies

```
void test(char *s) {  
    push canary;  
    char buf[10];  
    strcpy(buf, s);  
    ...  
    if (canary is changed) {log; exit;};  
}
```

- Ideia: canários nas minas
- Técnica em tempo de compilação
- Canário = 32bit random value
- Aparece no StackGuard; /GS flag in Visual Studio 7.0 and above
- Implementado pelo compilador

Stack smashing:



▶ 15

SS - Nuno Santos

2019



Canários / Stack cookies (cont.)

▶ Detecta alguns ataques stack smashing

- ▶ Ataques stack smashing que reescrevem o return address (EIP salvo) – para saltar para o código injectado
- ▶ Erros que reescrevem o EBP salvo

▶ Pode não detectar ataques BO que modificam variáveis locais (as que estão abaixo do canário)

▶ Detecta, mas possivelmente demasiado tarde, ataques BO contra argumentos das funções (que estão acima do return address)

▶ 16

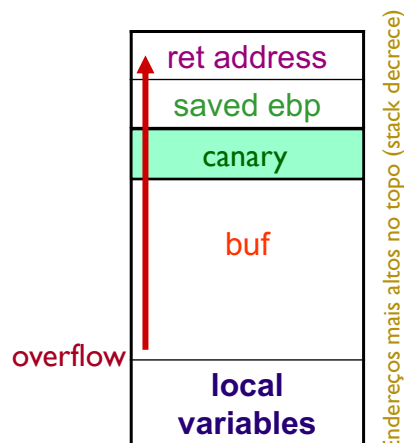
SS - Nuno Santos

2019



Protecção das variáveis locais

- ▶ Canários não protegem variáveis locais de overflows
- ▶ Solução: reordenar a ordem do stack layout
 - ▶ Todos os char buffers são colocados acima de todas as outras variáveis



▶ 17

SS - Nuno Santos

2019



Stack não executável

- ▶ Muitos ataques buffer overflow envolvem injectar código shell dentro da pilha
 - ▶ Protecção simples: marcar essa memória como páginas não-executáveis (NX)
- ▶ Vários nomes:
 - ▶ Intel – Execute Disable (XD-bit)
 - ▶ AMD – Enhanced Virus Protection
 - ▶ Microsoft – Data Execution Prevention (DEP)
 - ▶ Outros chamam-lhe W^X (*write or execute, but not both*)
- ▶ **Activado pelo compilador** (e.g., -z noexecstack no gcc); normalmente activado por omissão

▶ 18

SS - Nuno Santos

2019



Stack não executável (cont.)

▶ Limitações do NX

- ▶ Não protege contra *ret-to-libc/return-oriented programming attacks*
- ▶ Pode haver funções de biblioteca que podem ser chamadas para desactivar o bit NX
- ▶ Algumas aplicações podem ser incompatíveis com NX
 - ▶ Certas aplicações que fazem high-performance computing por vezes criam código binário em runtime
 - ▶ Algumas linguagens interpretadas compilam scripts em código binário

▶ 19

SS - Nuno Santos

2019



Validação de limites de arrays

- ▶ BOs são causados por falta de valiação dos limites dos arrays, pelo que fazer isto resolve o problema
- ▶ Já feito em linguagens como o Java
- ▶ Suportado hoje em compiladores C como gcc (flag *Warray-bounds*)
- ▶ No entanto, é fácil validar `a[3]`, mas não `*(a+3)` or `a[i]`

gcc dá warning	gcc não dá warning
<pre>char str[10]; int i; for (i=0; i<10; i++) { str[i] = 'a'; } str[10]='\0';</pre>	<pre>char str[10]; int i; for (i=0; i<10; i++) { str[i] = 'a'; } str[i]='\0';</pre>

▶ 20

SS - Nuno Santos

2019



Conclusões

- ▶ Duas técnicas adicionais que permitem mitigar vulnerabilidades de segurança são: análise estática e protecção dinâmica
- ▶ Análise estática tem por objectivo encontrar chamadas de funções perigosas, e conseguem ser bastante eficazes, sendo a maior desvantagem a grande quantidade de falsos positivos
- ▶ As técnicas de protecção dinâmica são implementadas em runtime e são bastante eficazes em evitar muitos ataques que exploram buffer overflows



Referências e próxima aula

- ▶ **Bibliografia**
 - ▶ [Correia17] Capítulos 14 e 15
- ▶ **Próxima aula**
 - ▶ Desenvolvimento de Software Seguro