

Investigando o Kernel com SystemTap

Fábio Olivé
<fabio.olive@gmail.com>



Sobre a Apresentação

- Nível: Intermediário a Avançado
- Escopo: apresentação da linguagem, técnicas, conceitos e exemplos de depuração do kernel com systemtap
- Pré-requisitos:
 - Compreensão do papel e funcionamento do kernel
 - Compreensão de código em C
 - Experiência com sistemas Unix e text-utils



Depuração de Kernel

- Instrumentação de código
 - `printk("O valor é %d\n", foo->bar);`
 - Arquivo no `/proc`
 - Ferramentas especiais (NETLINK?)
 - Edita -> Compila -> Testa -> Repete
 - crash
 - Consulta estado imediato do sistema
 - Analisa `crashdump`



Depuração de Kernel

- Necessidades para prover suporte adequado:
 - Eliminar edita/compila/testa/repete
 - Coletar estado do sistema em pontos variados
 - Tolerar pequenas diferenças entre versões
 - Poder enviar algo confiável para um cliente
 - Não piorar a situação
 - Um módulo de depuração quebrar o sistema fica mais feio do que o bug original!

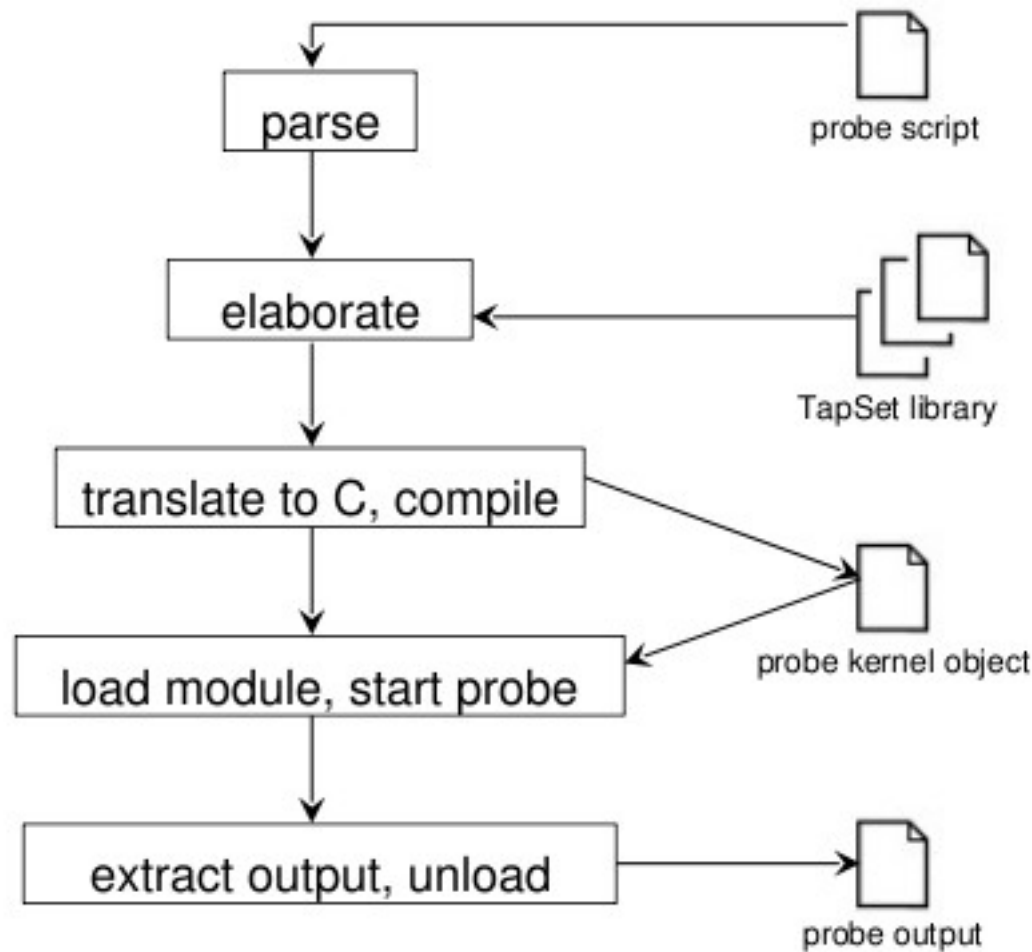


Depuração de Kernel

- Systemtaps!
 - Assemelham-se a breakpoints em um depurador, porém automatizados
 - Feitos em uma linguagem de alto nível, que é traduzida para C na hora de inserir no kernel
 - Podem interceptar quase qualquer função
 - Especializa-se em coletar informações de forma rápida e prática, sem efeitos colaterais

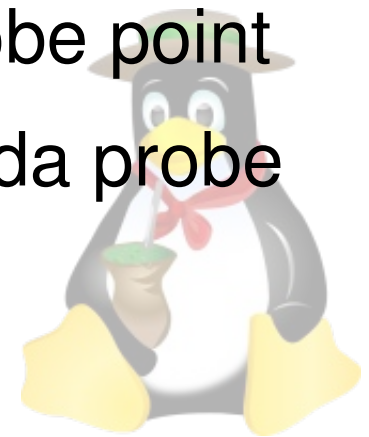


Ciclo de vida de uma Systemtap



Systemtaps

- Um systemtap é uma coleção de:
 - Probe points
 - Blocos de código para executar quando um probe point é executado
- Se assemelha a programação reativa, ou baseada em eventos
 - Eventos são: o processador atingiu o probe point
 - A reação ao evento é executar o código da probe



Probe points

- Definem pontos no código do kernel
 - Pontos de entrada de função (call probes)
 - Pontos de saída de função (return probes)
 - Eventos abstratos como timers, início ou fim de uma sessão de systemtap
 - Pontos quaisquer dentro dos arquivos fontes
 - Funções “nfs3_*@fs/nfs/nfs3proc.c”
 - Linha específica “@fs/nfs/nfs3proc.c:1234”
 - PERIGO!



Probe points

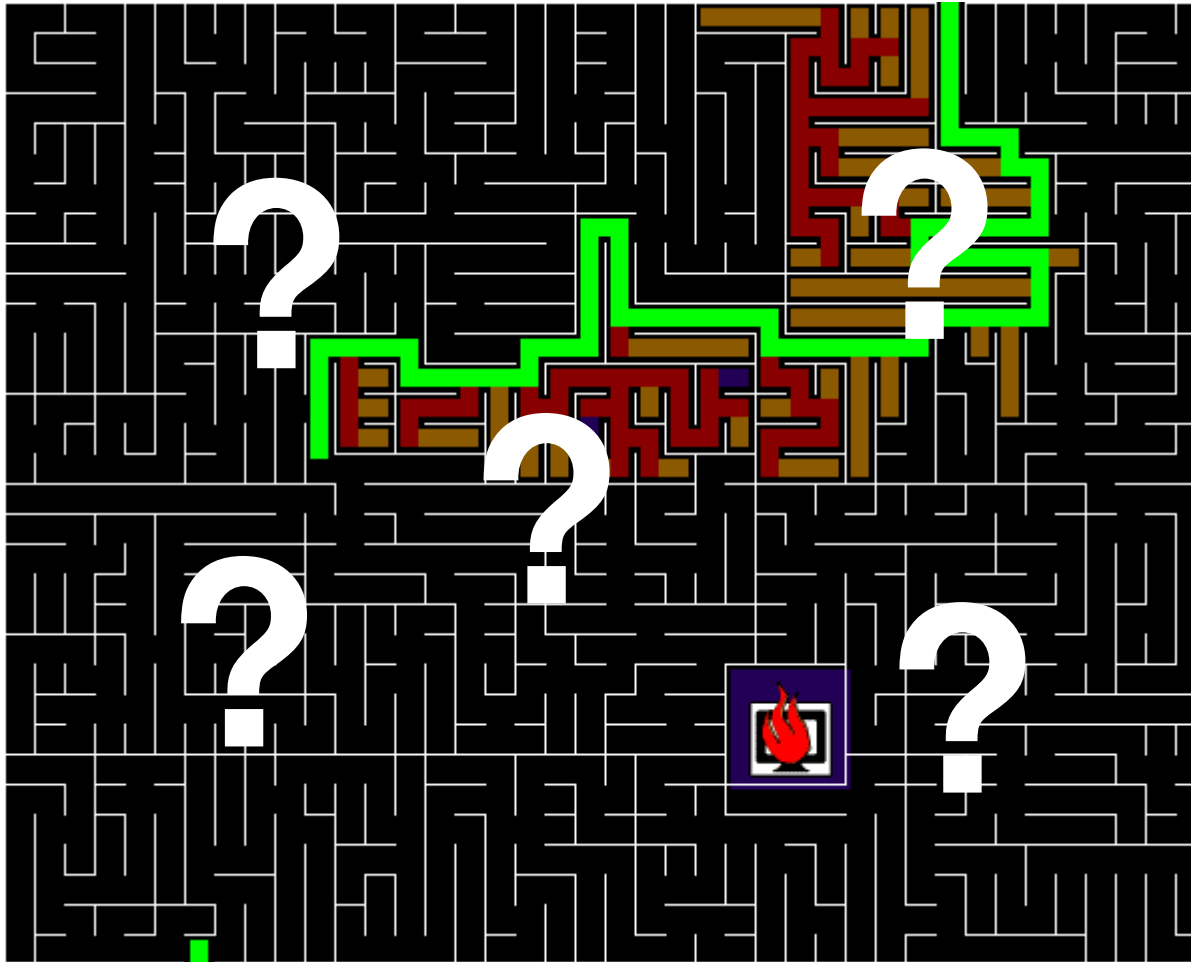
- Problema com probes mais esotéricas:
 - Código otimizado!
 - Módulos não carregados
 - Funções inline
 - Símbolos redefinidos ou macros que parecem funções
 - Código otimizado!



Código Otimizado

```
int foobar_baz(int foo, struct inode *bar)
```

```
{
```

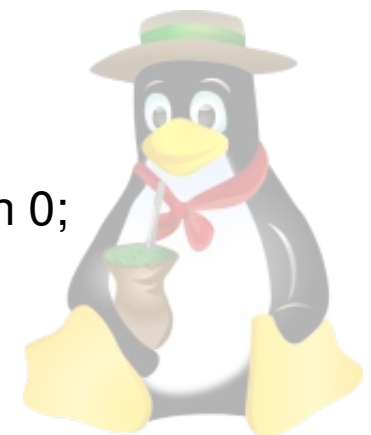


```
return -EINVAL;
```

```
return -EPERM;
```

```
return -EISDIR;
```

```
return 0;
```



Código Otimizado

- Como o kernel é otimizado, algumas variáveis podem sumir, o código pode ser reorganizado ou dividido em pedaços
 - Probes de linhas específicas podem não funcionar
- O mais garantido é usar probes em pontos síncronos
 - Call, return, timer, begin, end
 - Tracepoints! Markers!



Corpo das probes

- Linguagem parecida com AWK (e C)
 - Na verdade o conceito todo é parecido
- AWK: /padrão/ { código reativo; }
- STAP: probe kernel.function() { código reativo; }
- Variáveis inteiras ou string
 - Nada de ponto flutuante no kernel
- Operadores típicos, printf, etc



Corpo das probes

- Arrays indexados por qualquer coisa (hashes)
 - Inclusive múltiplos índices
- Variáveis globais (pra probe)
- Variáveis de “estatísticas”
 - Acumuladores rápidos de números
 - Pode-se pedir visões estatísticas interessantes
 - É bem frequente, portanto foi incluído na linguagem de forma otimizada



Corpo das probes

- Variáveis aparecem quando são mencionadas
 - Globals são declaradas
- Variáveis numéricas são sempre s64
 - Signed long long (inteiro de 64bits com sinal)
- Variáveis string podem ser user ou kernel
 - Sempre se usa `user_string()` (nunca consegui fazer funcionar) ou `kernel_string()` para referenciar o `char*` original do código



Runtime library

- Existem várias funções úteis que não precisam ser repetidas em todas as staps
 - man stapfuncs
 - Timestamps, identificação do processo atual, nível de indentação, etc
 - printf!
- O módulo fica enorme, mas a stap fica pequena ;-)



Exemplos

```
probe begin  
{  
    printf("Probe begins... \n")  
}
```

```
probe end  
{  
    printf("Probe ends... \n")  
}
```

```
probe timer.s(10)  
{  
    exit()  
}
```



Exemplos

```
# sys_open
#   do_sys_open
#     do_filp_open

probe kernel.function("do_filp_open")
{
    printf("%s(%d): opened %s\n",
          execname(),
          pid(),
          kernel_string($pathname))
}
```



Exemplos

global count

```
probe kernel.function("do_filp_open")  
{  
    count[pid()]++  
}
```

```
probe kernel.function("do_exit")  
{  
    printf("%s(%d) abriu %d arquivos\n",  
        execname(), pid(), count[pid()])  
    delete count[pid()]  
}
```



Técnicas Típicas

- Registrar (printf) informações quando uma função é chamada
 - Simplesmente ser chamada é relevante
- Registrar quando uma função retorna erro
 - Return probes
- Coletar informações e registrar:
 - Periodicamente
 - No final da sessão de stap



Cuidados

- Qualquer processo em qualquer CPU pode chamar a função sendo interceptada
 - Concorrência, globals
- Coletar informações em arrays indexados por PID
- Coletar informações genéricas em arrays indexados por nome de função
 - `probefunc()`



Instalando no Fedora

- Instalar o pacote `systemtap` (`yum install`)
- Habilitar repositórios `debuginfo` no `yum.repos.d`
 - Todos os pacotes são gerados com `debuginfo`
 - Todos os repositórios possuem uma seção `debuginfo`, porém desabilitada
- Instalar `kernel-debuginfo`
 - `yum install kernel-debuginfo`
- Opcionalmente, instalar os fontes do kernel em uso



Mais Informações

- sourceware.org/systemtap
- `man stap`
- `man stapfuncs`
- `man stapprobes`
- ... google? :)



Demonstração

Vamos pro terminal!



Dúvidas?

A única pergunta boba é a não perguntada.

