FACULDADE DE INFORMÁTICA E ADMINISTRAÇÃO PAULISTA - FIAP

CURSO DE TECNOLOGIA EM DEFESA CIBERNÉTICA

TRADUÇÃO CAPÍTULO 7

THE HARDWARE HACKER HANDBOOK

SÃO PAULO – SP

MAIO/2024

EMILY KAROLINE GONZAGA DOS SANTOS

TRADUÇÃO

THE HARDWARE HACKER HANDBOOK

ALEXANDRA PERCARIO

SÃO PAULO – SP

MAIO/2024

**7 X MARKS THE SPOT: TREZOR ONE WALLET MEMORY DUMP**

Vamos concluir esta série de capítulos sobre injeção de falhas quebrando um alvo real: a carteira Trezor One. Usaremos injeção de falha eletromagnética para demonstrar o despejo de memória e nos permitir extrair a semente de recuperação, que é tudo o que é necessário para acessar o conteúdo da carteira.

Este capítulo será o mais aberto do livro. Ele descreve um ataque avançado que pode exigir equipamentos mais especializados e tem uma taxa de sucesso muito baixa, mesmo quando bem ajustado. Na verdade, recriar este ataque seria um bom projeto de semestre acadêmico. Para acompanhar todo o ataque, você precisará de um entendimento sólido de design embarcado, juntamente com uma configuração de instrumentação complicada e um pouco de sorte por cima. No entanto, achamos importante mostrar o que é necessário para passar de dispositivos simples para produtos reais.

Discutimos a injeção de falha eletromagnética (EMFI) na seção "Injeção de Falha Eletromagnética" na página XX. A EMFI tenta construir um pulso poderoso imediatamente acima da superfície superior do próprio dispositivo, causando todos os tipos de corrupção dentro do alvo. Neste capítulo, usaremos uma ferramenta EMFI chamada ChipSHOUTER para realizar a injeção.

**Introdução ao Ataque**

Nosso alvo é uma carteira de bitcoins Trezor One. Este pequeno dispositivo pode ser usado para armazenar bitcoins, o que significa, em última análise, que ele fornece um método de armazenamento seguro de uma chave privada usada para operações criptográficas. Não precisamos nos aprofundar nos detalhes da operação da carteira, mas entender a ideia de uma semente de recuperação é crítico. A semente de recuperação é uma série de palavras que codificam uma chave de recuperação, e saber essa semente de recuperação é suficiente para recuperar a chave privada. Isso significa que alguém que rouba apenas a semente de recuperação (sem acesso adicional à carteira) poderia acessar os fundos armazenados na própria carteira. Um ataque que encontre a chave seria bastante prejudicial para a segurança das preciosas moedas do proprietário.

O ataque que descrevemos aqui foi inspirado em alguns outros trabalhos. A apresentação "wallet.fail" no Chaos Computer Club (CCC) por Dmitry Nedospasov, Thomas Roth e Josh Datko demonstrou como quebrar a proteção de segurança STM32F2 e despejar o conteúdo da RAM estática (SRAM). Em vez disso, mostraremos como despejar diretamente o conteúdo da memória flash onde a semente está armazenada, então é um ataque diferente, mas com resultados semelhantes. Usaremos EMFI, permitindo-nos realizar o ataque sem sequer remover a caixa. Isso significa que alguém pode realizar o ataque sem deixar nenhum vestígio de modificação na carteira, não importa o quão cuidadosamente ela seja inspecionada. Este capítulo apresenta várias ferramentas mais avançadas, e você verá em seu uso que pode valer a pena o investimento quando se trata de examinar alvos reais. Como exemplo, usaremos USB como uma forma de cronometrar nosso ataque. Um verdadeiro espião USB (como um Total Phase Beagle USB 480) é fundamental aqui para entender essa cronometragem. Temos uma discussão mais longa sobre ferramentas no Apêndice A.

**Nota:** O ataque neste capítulo, primeiro descrito por Colin (coautor deste livro) como parte do artigo "MIN()imum Failure: EMFI Attacks against USB Stacks", foi apresentado no Workshop USENIX sobre Tecnologia Ofensiva (WOOT) em 2019.

**Interior do Wallet Trezor One**

A carteira Trezor One é de código aberto, o que torna este ataque uma demonstração maravilhosa para ensinar EMFI e injeção de falhas. Você pode modificar livremente o código ou programar versões mais antigas que ainda não corrigiram a vulnerabilidade.

As fontes do Trezor estão disponíveis no GitHub no projeto trezor-mcu. Se você quiser seguir os passos deste capítulo, selecione a tag "v1.7.3" no GitHub, ou siga o link https://github.com/trezor/trezor-mcu/tree/v1.7.3, que o levará a esta versão exata. Essas falhas foram corrigidas há muito tempo em uma versão do firmware que estará disponível quando você ler este livro, então você precisará olhar o código mais antigo (vulnerável) para entender melhor o ataque exato. O Trezor é baseado em um STM32F205. A Figura 7-1 mostra o dispositivo sem a caixa.



Os seis soquetes de pinos no lado esquerdo da placa de circuito impresso (PCB) são o cabeçalho JTAG. O STM32F205 está logo abaixo da superfície da caixa, um recurso que usaremos para tornar nosso ataque mais realista em cenários práticos.

A semente de recuperação sensível real está armazenada na memória flash em uma seção chamada metadados. Está localizado logo após o bootloader, como mostrado no Exemplo 7-1. Parte do arquivo de cabeçalho define a localização de vários itens de interesse dentro do espaço da memória flash.

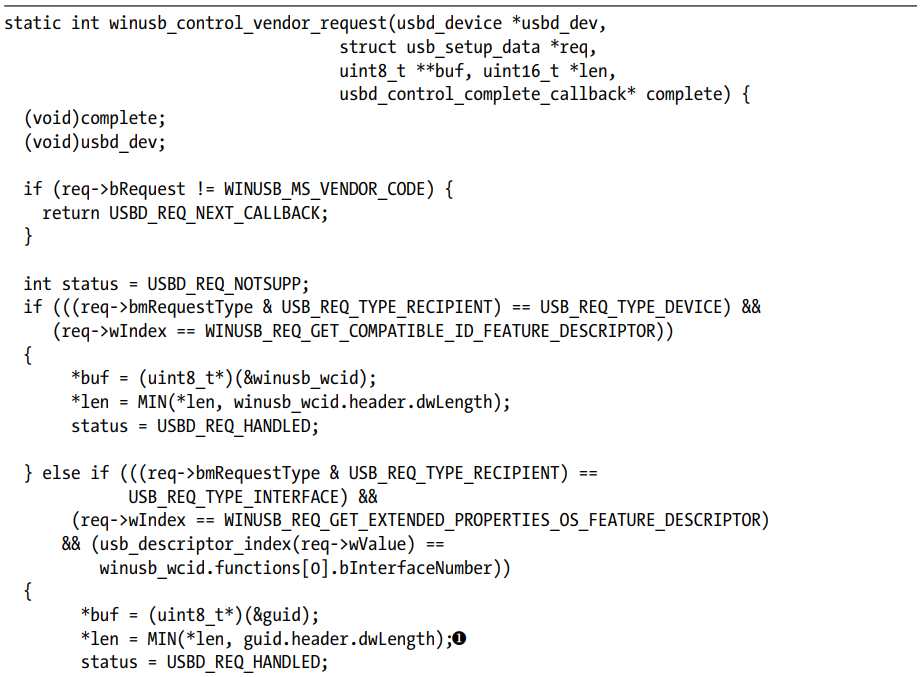


O endereço FLASH\_META\_START está no final da seção do bootloader. Você pode entrar no bootloader pressionando os dois botões na frente do Trezor, o que permite que uma atualização de firmware seja carregada via USB. Como uma atualização de firmware maliciosa poderia simplesmente ler os metadados, o bootloader verifica se várias assinaturas estão presentes em uma atualização de firmware para evitar tal ataque. Usar injeção de falha para carregar firmware não verificado seria um método de ataque, mas não é o que vamos usar. O problema com todos esses ataques é que o Trezor apaga a memória flash antes de carregar e validar o novo arquivo, armazenando os metadados sensíveis na SRAM durante esse processo. A divulgação da wallet.fail realmente atacou esse processo, pois é possível falhar o STM32 para ir do nível de proteção de leitura de código RDP2 (que desativa completamente o JTAG) para o nível RDP1 (que permite que o JTAG leia da SRAM, mas não do código).

Se nosso ataque corrompesse a SRAM (ou precisasse de um ciclo de energia para se recuperar de estados de erro), realizar esse apagamento seria muito perigoso. O ataque da wallet.fail conseguiu recuperar a SRAM, mas o método de ataque que usaremos pode corromper a SRAM, o que significa que qualquer erro destruiria permanentemente a semente de recuperação. Em vez disso, tentaremos ler diretamente a memória flash, o que é muito mais seguro, pois garantimos que um comando de apagamento não será executado, o que significa que os dados são armazenados com segurança na memória, aguardando nossa extração.

**Falha na Solicitação de Leitura USB**

Como o bootloader suporta USB, ele também contém código de processamento USB muito padrão. O Exemplo 7-2 mostra parte dele, que vem do arquivo winusb.c na árvore de código-fonte do firmware do Trezor. Escolhemos esta função de "solicitação de fornecedor de controle" em particular porque ela envia o "guid" através do USB.





A função de solicitação de controle primeiro verifica algumas informações enviadas sobre a solicitação USB. Ele procura por um bRequest correspondente, bmRequestType e wIndex, que são todos atributos de uma solicitação USB. Finalmente, a própria solicitação USB contém um campo wLength, que é quanto dados o computador está solicitando que seja enviado de volta. Isso é passado para a função do Exemplo 7-2 como o argumento \*len. (O observador cuidadoso também notará o membro de estrutura dwLength no Exemplo 7-2, que tem uma função completamente diferente: dwLength é o tamanho dos dados disponíveis para enviar de volta com base no descritor programado no dispositivo.) Podemos solicitar livremente até 0xFFFF bytes de dados, e é exatamente isso que faremos. No entanto, o código realiza uma operação MIN() para limitar o comprimento dos dados reais enviados de volta ao computador ao mínimo entre o comprimento solicitado ou o tamanho do descritor que enviaremos de volta. O computador sempre pode solicitar uma quantidade menor de dados do que o tamanho do descritor, mas se solicitar mais dados do que o dispositivo tem (ou seja, se solicitar um tamanho de resposta maior do que o comprimento do descritor), o dispositivo simplesmente envia de volta apenas os dados válidos.

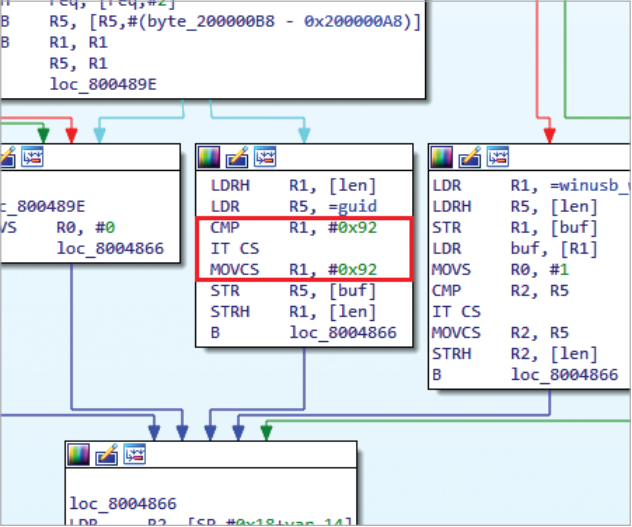
O que acontece se essa chamada MIN() no wLength retornar o valor errado? Enquanto o código responderia com o descritor (como esperado), também enviaria todos os dados após o descritor até o deslocamento 0xFFFF a partir do início do descritor. Isso acontece porque a chamada MIN() está garantindo que a solicitação do usuário permita apenas a leitura da memória válida de volta, mas se a chamada MIN() retornar o valor errado, isso significa agora que a solicitação do usuário pode ler mais do que a memória antecipada. Esta seção de memória "mais do que o antecipado" inclui nossos preciosos metadados. A pilha USB não sabe que os dados não devem ser enviados de volta. A pilha USB está simplesmente enviando de volta o bloco de dados conforme solicitado pelo computador. Toda a segurança do sistema depende de uma simples verificação de comprimento.

Aqui está nosso plano: Usaremos injeção de falha para ignorar a verificação que depende de uma única instrução. Aproveitamos o fato de que o bootloader (e o "guid") está localizado em um endereço mais baixo na memória do que onde está armazenada a semente de recuperação sensível. Estamos planejando despejar a memória lendo de um endereço mais baixo para um endereço mais alto, então o ataque provavelmente terá sucesso apenas quando atacarmos o código USB no bootloader. Se atacarmos o código USB na aplicação regular que vive em FLASH\_APP\_START, é mais provável que as partes interessantes já apontem além da área sensível FLASH\_META\_START.

Antes de nos aprofundarmos nos detalhes de realizar a falha real, vamos fazer um pouco de verificação de sanidade de nossas alegações. Você pode usar tais verificações em seu próprio código para ajudar a entender o impacto de vulnerabilidades semelhantes.

**Desmontando o Código**

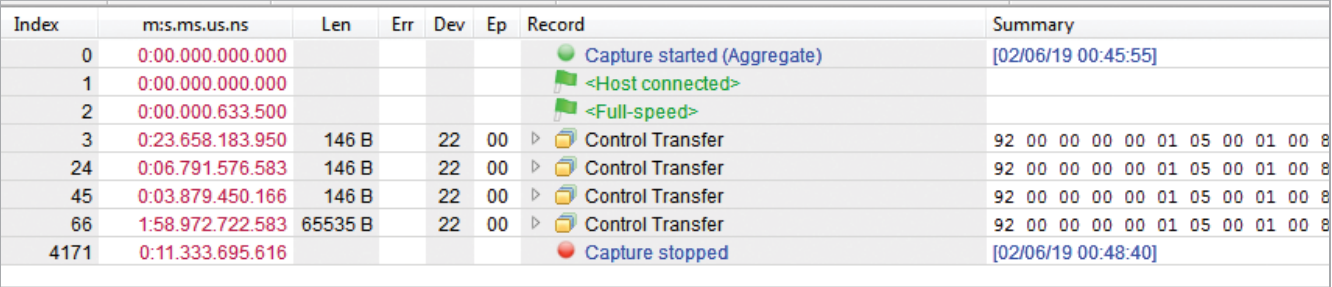
A primeira verificação de sanidade é confirmar que uma falha simples pode causar nossa operação pretendida. Podemos facilmente fazer isso inspecionando uma desmontagem do firmware do Trezor em execução no dispositivo usando o Interactive Disassembler (IDA), que exibe uma quebra de código de montagem (do Exemplo 7-2), como mostrado na Figura 7-2.



O valor de entrada de wLength foi armazenado em R1, e R1 é comparado com 0x92 na desmontagem. Se for maior, é definido como 0x92 com uma movimentação condicional (MOVCS em assembly ARM). Essas linhas de montagem são a implementação da chamada MIN(\*len, guid.header.dwLength) no código C do Exemplo 7-2. Devido ao fluxo de código resultante que podemos observar na desmontagem, precisamos pular apenas a instrução MOVCS para realizar nosso objetivo de ter o campo wLength fornecido pelo usuário ser aceito.

A segunda verificação de sanidade é confirmar que nenhuma proteção de camada superior existe. Por exemplo, talvez a pilha USB na verdade não aceite uma resposta tão grande, já que não há um requisito real para isso. Confirmar isso é um pouco mais difícil de fazer por inspeção simples, mas a natureza de código aberto do Trezor torna isso possível. Podemos simplesmente modificar o código para comentar a verificação de segurança e, em seguida, verificar se podemos solicitar uma grande quantidade de memória. Se você não quiser recompilar o código mas tiver acesso ao depurador, também poderia usar um depurador anexado para definir um ponto de interrupção no MOVCS e alternar o status da flag ou manipular o contador de programa para ignorar a instrução.

Validar essa verificação de sanidade é feito da mesma maneira que o ataque real. Vamos trabalhar todos os detalhes nas seções a seguir. Por enquanto, apenas mostraremos como não existem outros obstáculos para obter um grande buffer por meio da solicitação de controle. O código de ataque envia uma solicitação de comprimento de 0xFFFF para a solicitação. A Figura 7-3 mostra o tráfego USB capturado com o Total Phase Beagle USB 480. Quando não modificamos a instrução MOVCS, a solicitação USB resulta no comprimento esperado de 146 (0x92) bytes, mostrado nos índices 3, 24 e 45.



Modificar a instrução (ou usar um depurador para limpar manualmente a flag de comparação) para ignorar esta verificação resulta em uma resposta de tamanho completo, já que o comprimento do índice 66 é 65535, ou 0xFFFF. Isso demonstra que não existe nenhum recurso oculto que impedirá fundamentalmente que o ataque funcione.

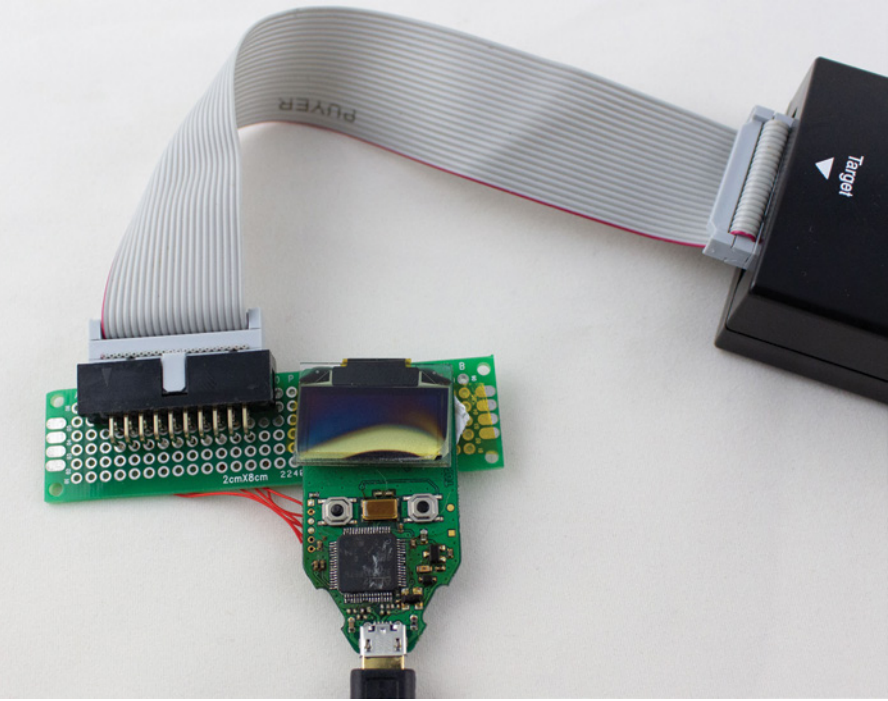
**Desenvolvimento de Firmware e Validação da Falha**

Basicamente, seguiremos a documentação para compilar o firmware do Trezor do Guia do Desenvolvedor do Trezor disponível na Wiki do Trezor online. Aqui estão os passos específicos:

1. 1. Clonar o firmware de produção e verificar uma revisão conhecida como vulnerável.
2. 2. Compilar o firmware sem proteção de memória.
3. 3. Programar e testar o dispositivo.
4. 4. Editar o firmware para remover a verificação de comprimento USB e tentar nosso ataque.

**Aviso:** Para seguir os passos, você precisará de um dispositivo Trezor no qual possa carregar seu próprio bootloader. Dispositivos Trezor de produção não permitem que você reprogramar o bootloader com versões não assinadas por razões de segurança e têm o JTAG desativado, mesmo se você usar um programador externo. Você precisará de um Trezor onde tenha substituído o STM32F205RGT6 por um chip de substituição em branco ou uma placa de desenvolvimento compatível com o Trezor. Consulte a Wiki do Trezor para mais informações.

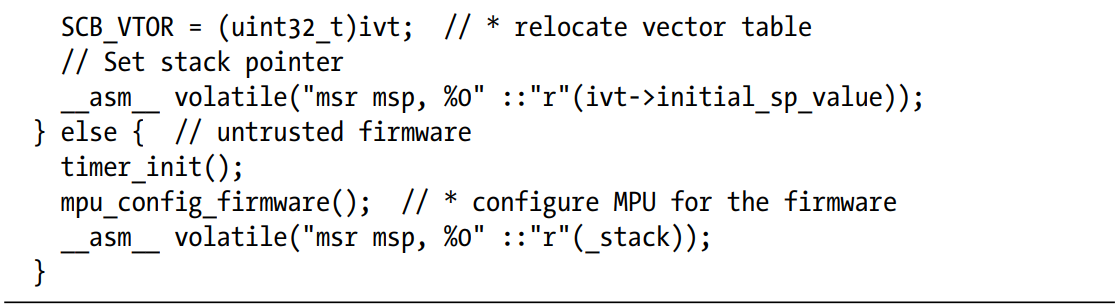
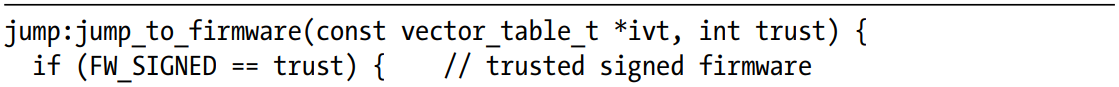
A Figura 7-4 mostra o Trezor com um depurador JTAG anexado. Este Trezor é uma unidade de produção com o chip principal substituído.



Usamos um SEGGER J-Link para o depurador, mas um ST-Link/V2 também funcionaria e custa muito menos. O esquemático da placa Trezor está disponível no repositório do GitHub do hardware Trezor, https://github.com/trezor/trezorhardware/tree/master/electronics/trezor\_one, que detalha o mapeamento dos pontos de teste na placa.

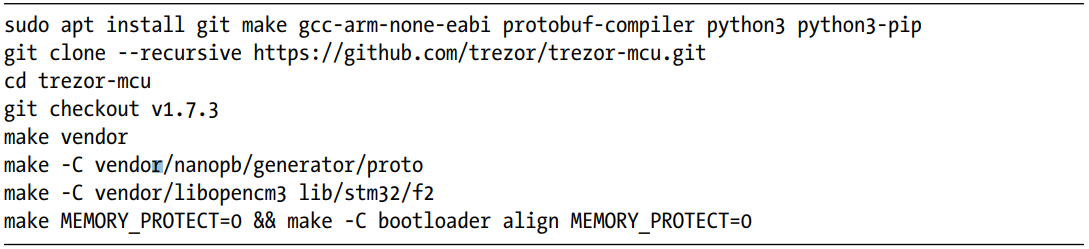
**Nota:** Você poderia usar a divulgação do wallet.fail para desbloquear o JTAG e apagar o dispositivo também, se realmente quiser ser elite. Se não quiser validar a falha em simulação, tente aplicar a falha diretamente em uma versão de produção do firmware 1.7.3. Use o utilitário de linha de comando trezorctl para carregar uma versão específica do firmware no dispositivo com o comando trezorctl firmware-update -v 1.7.3. Você deve ver a tela indicar que "Loader 1.6.1" está em execução, onde 1.6.1 é a versão do bootloader que foi enviado com o firmware principal 1.7.3. Você precisa ter exatamente essa versão para que esse ataque funcione.

Como qualquer firmware que construirmos dessa forma será não assinado, o Trezor bloqueará nossa capacidade de reprogramar o bootloader a partir do firmware não assinado. Isso significa que construir completamente o firmware final é inútil, já que isso significaria que precisaríamos reescrever o bootloader. A Listagem 7-3 mostra uma seção do código que protege o bootloader.



Se um firmware não confiável for carregado, a unidade de proteção de memória é configurada para desabilitar o acesso à seção do bootloader da memória flash. Se o código na Listagem 7-3 não estivesse lá, poderíamos ter usado um código de aplicativo personalizado para carregar o bootloader que queremos avaliar.

Os primeiros passos para construir o bootloader são fáceis (veja a Listagem 7-4) e seguem aproximadamente a documentação. Você precisará fazer isso em um computador Linux ou em uma máquina virtual Linux; nossos exemplos são no Ubuntu. Vamos construir apenas o próprio bootloader, já que é aí que está a vulnerabilidade. Essa sequência de compilação evita algumas dependências para construir a aplicação completa (principalmente protobuf) que podem ser um pouco mais difíceis de instalar.



Você pode precisar fazer ajustes adicionais para que isso funcione. Dependendo do compilador, o bootloader pode ficar muito grande, nesse caso, exportar CFLAGS=-Os pode ajudar. Se isso funcionar, você produzirá um arquivo chamado bootloader/bootloader.elf.

A linha com MEMORY\_PROTECT=0 é crítica para a depuração. Se você errar a grafia (ou esquecer) dessa linha, alguma lógica de proteção de memória é habilitada. Uma coisa que a proteção de memória faz é bloquear o JTAG de modo que o uso futuro seja impossível. Para se poupar de futuros erros, recomendamos editar o arquivo memory.c e retornar imediatamente da função memory\_protect() na linha 30. Se você programar e executar o bootloader sem desabilitar a proteção de memória, perderá imediatamente a capacidade de reprogramar ou depurar o chip (permanentemente). Editar esse arquivo evitará que você fique muito infeliz quando precisar substituir o chip na sua placa.

O arquivo principal Makefile constrói uma pequena biblioteca, que inclui a lógica de proteção de memória. Para evitar esquecer de reconstruir a biblioteca, sugerimos executar os dois comandos em uma linha, conforme mostrado na Listagem 7-3. Isso também construirá o arquivo winusb.c que tem o código que queremos validar.

E agora? Agora você pode carregar o código de firmware construído usando um programador. Usamos um ST-Link/V2. Antes de programar o código, confirme novamente que você desabilitou o código de proteção de memória nesta compilação. Novamente, a Figura 7-4 mostra a conexão física do JTAG. Você precisará do software de programação para o ST-Link/V2; no Windows, este é o utilitário STM32 ST-LINK fornecido pela ST e, no Mac ou Linux, você pode construir o utilitário de código aberto stlink.

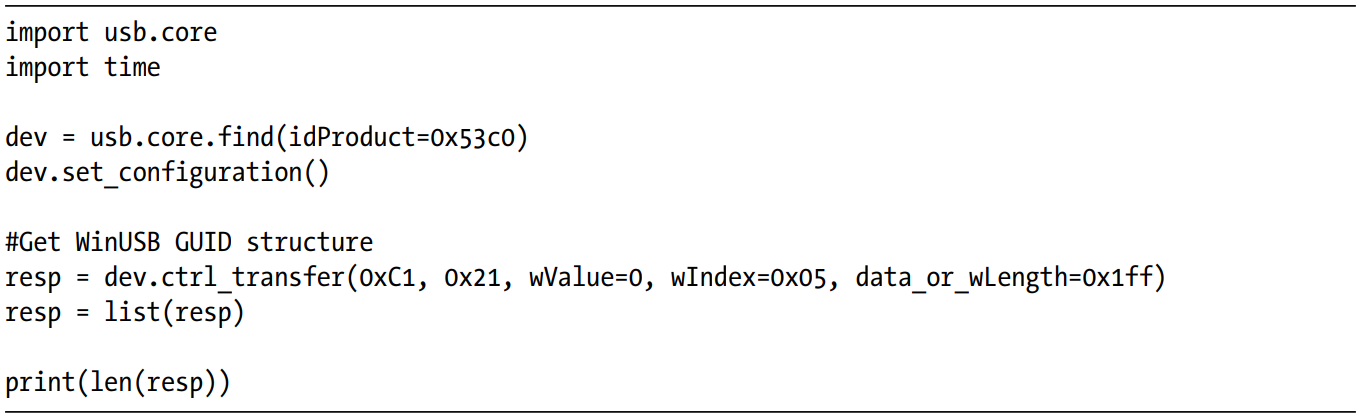
O próximo passo é manter o modo bootloader ativado e enviar algumas solicitações USB interessantes. Para fazer isso, conecte o dispositivo enquanto mantém pressionados os dois botões para entrar no modo bootloader. Se você estiver usando um dispositivo com um LCD (não é necessário para este experimento), verá o modo de bootloader listado.

Em seguida, você usará Python com PyUSB, que pode instalar com o comando pip install pyusb.

No Linux, você deve ser capaz de se comunicar diretamente com o dispositivo Trezor. O objetivo é executar o código Python na Listagem 7-5, que imprimirá que leu 146 bytes. Provavelmente, você precisará configurar as regras udev para o dispositivo Trezor (ou executar o script como root).

Usar um sistema semelhante ao Unix fornecerá os resultados mais confiáveis. O Windows frequentemente desativa uma porta USB se muitos eventos estranhos acontecerem nela, o que complica nossas tentativas de pesquisa.

A Listagem 7-5 assume que você está usando Linux.



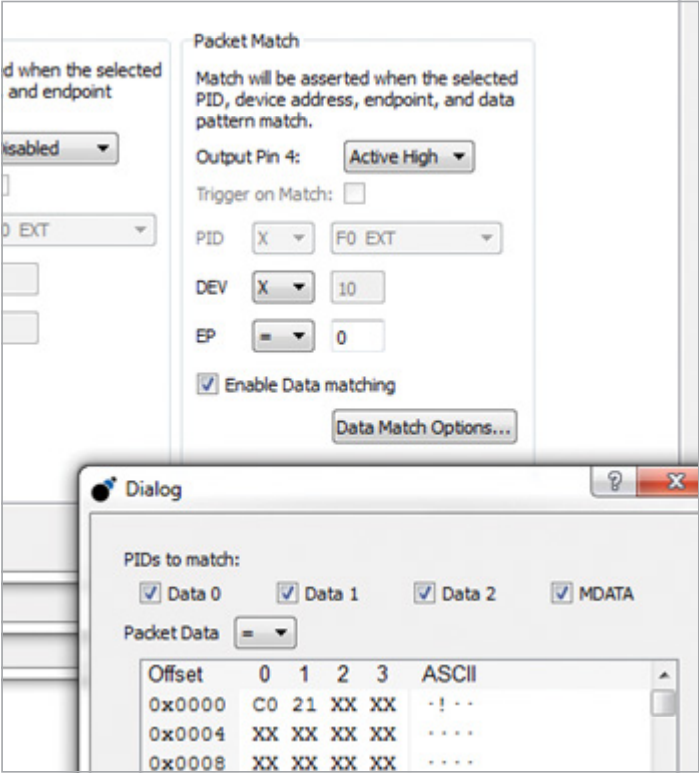
A variável data\_or\_wLength solicitou 0x1ff (511) bytes, mas apenas 146 devem ser retornados, pois esse é o comprimento do descritor. Experimente com quanto dados você pode solicitar. Você pode notar que em algum momento seu sistema operacional realmente retorna um "parâmetro inválido". Em teoria, em alguns sistemas, podemos solicitar até 0xFFFF bytes, mas muitos sistemas operacionais não permitem que você vá tão longe. Quando chegar a hora de fazer a falha, você desejará garantir que sua solicitação não seja interrompida pelo próprio sistema operacional, então encontre o limite superior da sua configuração.

Você também pode precisar aumentar o tempo limite para a chamada dev.ctrl\_transfer() na Listagem 7-5 adicionando o parâmetro timeout=50. As solicitações de controle normalmente retornam muito rapidamente, mas se você ler com sucesso grandes blocos de dados, os tempos limite padrão podem ser muito curtos.

**Desencadeamento USB e Cronometragem**

Antes de podermos inserir a falha, precisamos saber quando inseri-la. Sabemos a instrução exata que queremos que a falha atinja e conhecemos o comando que enviamos via USB. No entanto, precisamos fazer melhor do que isso para cronometrar a falha na instrução exata. No nosso caso, como temos acesso ao software, vamos "trapacear" durante nosso primeiro teste e medir o tempo real de execução. Se não tivéssemos essa capacidade, acabaríamos com um processo muito mais lento ou precisaríamos forçar o timing certo por tentativa e erro.

Primeiro, precisaremos obter um gatilho mais sólido nos dados USB em si. O método clássico para isso é usar algo como o Total Phase Beagle USB 480, que pode realizar o disparo com base nos dados físicos transmitidos pela linha USB. A Figura 7-5 mostra o setup.

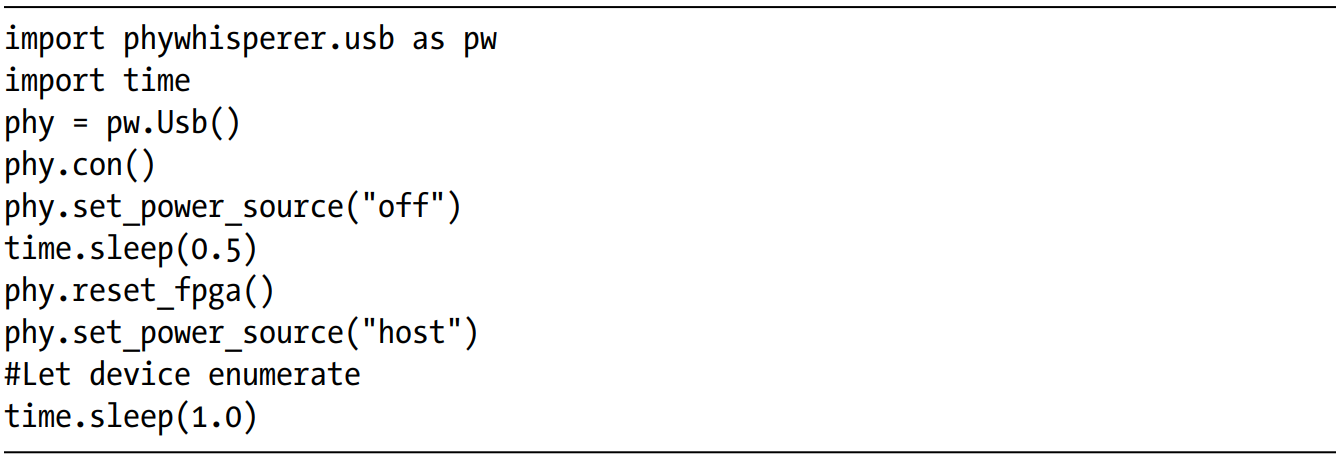


O Total Phase Beagle USB 480 também possui uma bela interface de sniffador, então podemos farejar o tráfego e entender melhor quais pacotes (malformados) estão sendo recebidos. Essa capacidade é muito útil, já que podemos ver, por exemplo, a parte exata da solicitação USB sendo interrompida/corrompida, o que pode fornecer algumas dicas sobre quão longe no código o programa foi executado.

Se você não tiver o Beagle, o Micah Scott desenvolveu um módulo simples para realizar falhas em tempo real chamado FaceWhisperer, que está disponível no GitHub. Ele usa USB para acionar a falha e foi usado com falhas de voltagem para despejar o firmware de uma mesa digitalizadora. Kate Temkin, da Great Scott Gadgets, também desenvolveu várias ferramentas, incluindo complementos para o GreatFET e várias ferramentas USB, como o LUNA. Usamos uma ferramenta que o Colin desenvolveu, o PhyWhisperer-USB de código aberto.

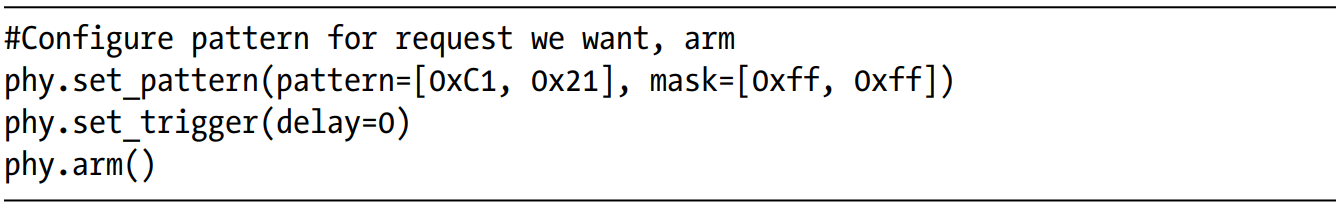
O PhyWhisperer-USB é projetado para realizar o desencadeamento USB com base em pacotes específicos. O USB do Trezor passa pelo PhyWhisperer-USB, de modo que um computador ainda está enviando as mensagens USB reais para o dispositivo Trezor.

O PhyWhisperer-USB é usado por meio de um programa Python (ou notebooks Jupyter). A Listagem 7-6 mostra a configuração inicial, que simplesmente se conecta ao PhyWhisperer-USB.



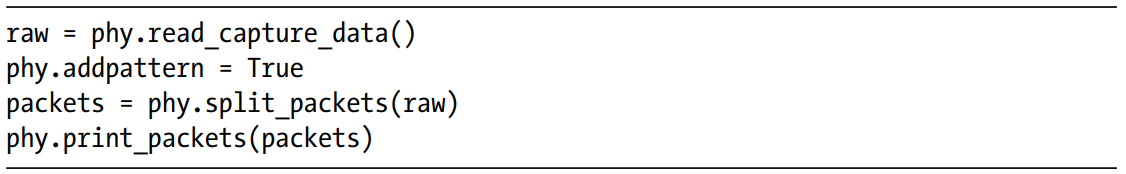
A configuração requer que você segure os botões no Trezor para garantir que ele inicie no modo bootloader. Este script reinicializa o alvo para que o PhyWhisperer-USB possa corresponder à velocidade USB observando a sequência de enumeração.

Toda vez que queremos um gatilho, configuramos o gatilho e armamos o PhyWhisperer-USB, conforme mostrado na Listagem 7-7.

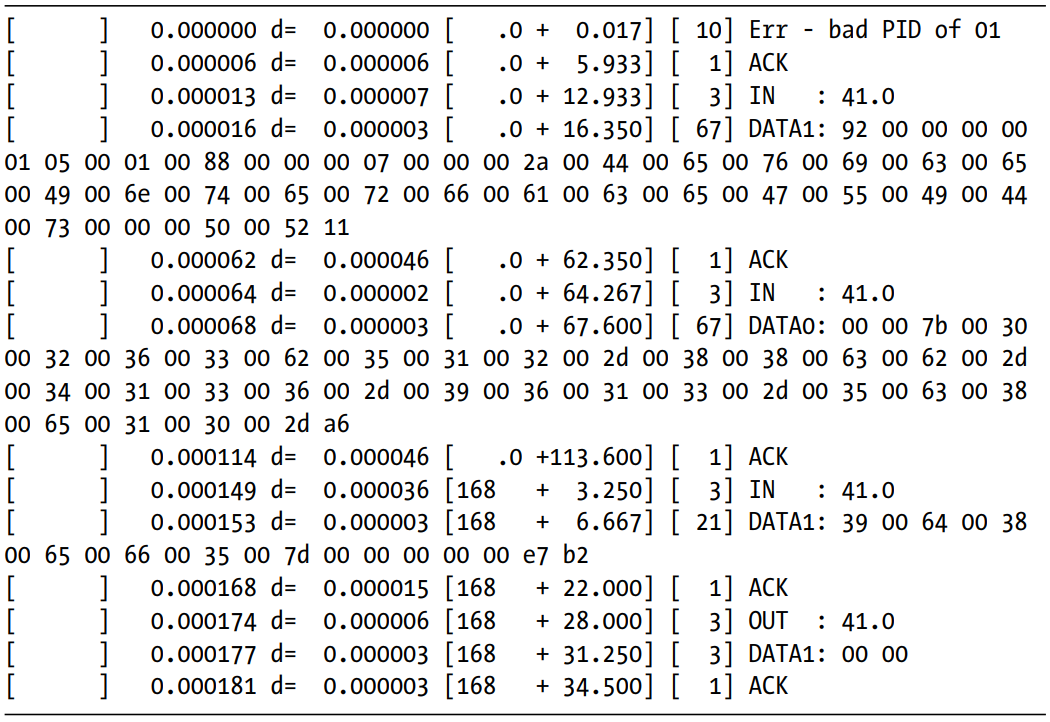
 Aqui definimos o gatilho com base na solicitação que estamos enviando (mostrada na Listagem 7-5). Podemos executar o código na Listagem 7-5 no sistema hospedeiro, que inicia o código que queremos falhar na Listagem 7-2 no Trezor. O conector Trig Out no PhyWhisperer-USB terá um pulso de gatilho curto que coincide com a solicitação USB passando pelo fio.

Mais tarde, durante o ataque de falha, usaremos o PhyWhisperer-USB para determinar o intervalo de tempo entre a solicitação USB e a instrução específica que queremos falhar. Depois que a solicitação USB aciona a execução do código, levará um pequeno período de tempo antes que a instrução de destino real seja executada. Ajustar os parâmetros de set\_trigger() nos permite alterar a saída do gatilho para um ponto posterior no tempo, a fim de alinhar o tempo da falha com a instrução de destino.

A vantagem do PhyWhisperer-USB é que também podemos monitorar o tráfego USB. A captura de dados USB começa com o gatilho; usamos o código na Listagem 7-8 para lê-lo do PhyWhisperer-USB.



A Listagem 7-9 mostra os resultados da captura, que são úteis para observar que os pacotes certos foram usados para o gatilho e se ocorreram erros USB.

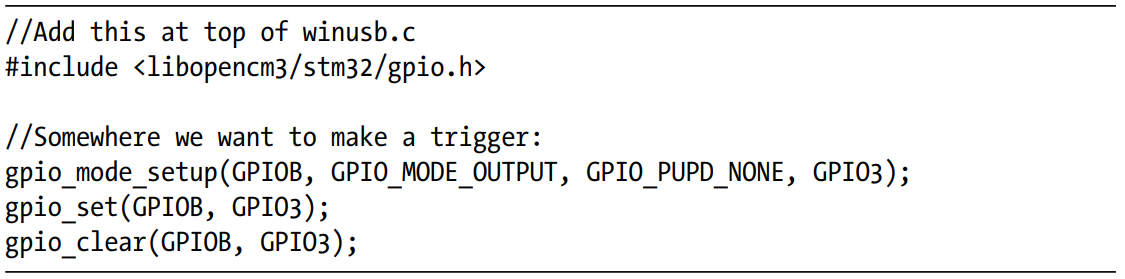


Observe o erro Err - bad PID de 01 na primeira linha devido à captura ter começado parcialmente durante um pacote de controle. Ajustar o padrão de gatilho para incluir o pacote completo evitaria este erro. Para nosso ataque aqui, este erro é irrelevante.

Ao automatizar nosso ataque de falha, podemos detectar falhas que não são o efeito desejado (ler dados em excesso), mas que ainda corrompem dados USB ou causam erros. Saber o tempo desses erros é informação útil. Se vemos um erro ocorrendo depois que já retornamos os dados USB, sabemos que nossa falha é muito tarde para ser eficaz, por exemplo.

Uma vez que temos um gatilho com base na solicitação USB passando "pelo fio", também inseriremos um segundo gatilho definindo um pino de E/S alto no Trezor quando o código sensível é executado. Usamos isso para caracterizar o timing, já que podemos usar um osciloscópio para medir o tempo desde a solicitação USB passando pelo fio até o momento da execução do código sensível.

Podemos encontrar um pino de E/S de reposição útil inspecionando o esquemático da placa Trezor; no nosso caso, encontramos o esquemático para v1.1 em https://github.com/trezor/trezor-hardware/blob/master/electronics/trezor\_one/trezor\_v1.1.sch.png. Vemos que o pino SWO do cabeçalho K2 (visível na Figura 7-1) é roteado para o pino de E/S PB3. Se o Trezor puder alternar PB3 durante a operação de comparação, isso fornecerá informações úteis de temporização para a realização de injeção de falhas. Isso nos poupa de ter que varrer um grande intervalo de tempo. A Listagem 7-10 mostra um exemplo simples de como realizar uma alternância de GPIO no STM32F215 no Trezor.



Se inserirmos o código na Listagem 7-10 no local em que queremos a falha, reconstruímos o bootloader e depois executamos o código, devemos obter um pulso curto no pino SWO que podemos usar para a temporização. Novamente, para realizar essa avaliação, você precisará de um Trezor que tenha sido hackeado para permitir a reprogramação.

Neste caso, o tempo entre o gatilho do PhyWhisperer-USB e o gatilho do Trezor acaba sendo em torno de 4,2 a 5,5 microssegundos. Não é uma temporização perfeita, já que parece haver alguma oscilação devido aos pacotes USB sendo processados por uma fila. Ver tal oscilação nos diz que, ao realizar a injeção de falhas, não devemos esperar alcançar uma confiabilidade perfeita. No entanto, nos dá uma faixa na qual podemos variar o parâmetro de temporização.

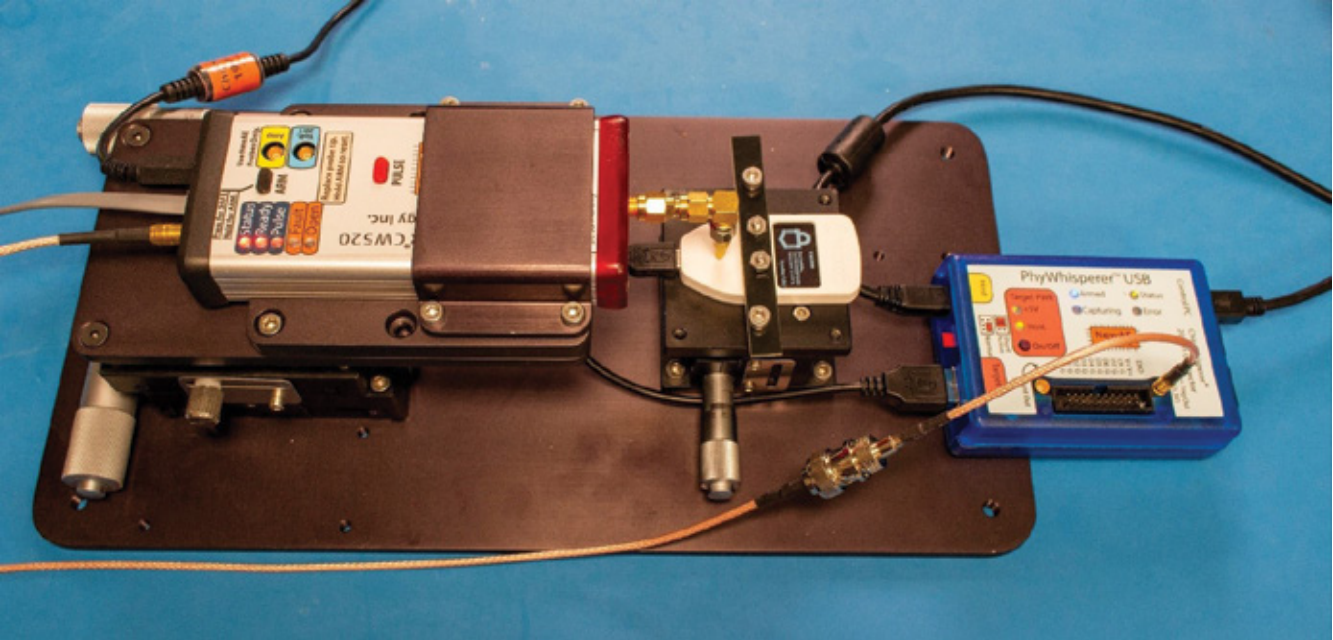
**Injeção de Falha através da Carcaça**

Nesta seção, passaremos da exploração do alvo para realmente falhar nele.

**O Setup**

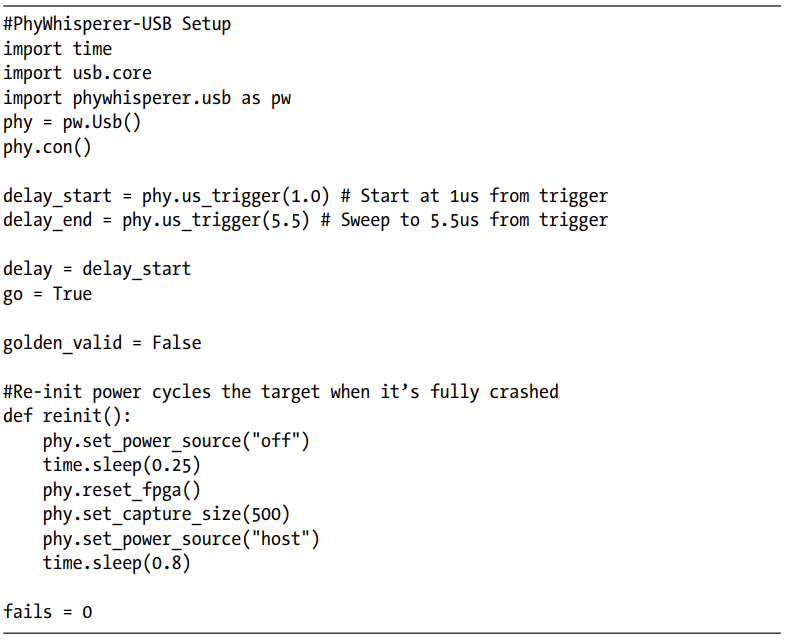
Para inserir a falha, nosso setup (mostrado na Figura 7-6) inclui uma ferramenta ChipSHOUTER EMFI montada em uma mesa XY manual para posicionar com precisão a bobina. O alvo Trezor também está montado em uma mesa XY, e o PhyWhisperer-USB fornece o acionamento e o controle de energia do alvo através de um interruptor dentro do PhyWhisperer-USB. A capacidade de controle de energia é útil, pois podemos redefinir o alvo quando ele travar. O controle de energia é um recurso comum em equipamentos específicos de injeção de falhas, mas ferramentas de propósito geral como o Beagle USB 480 estão ausentes.

O "gabarito" físico no qual o Trezor está montado pressiona os dois botões do painel frontal, garantindo que ele sempre entre no modo bootloader na inicialização.



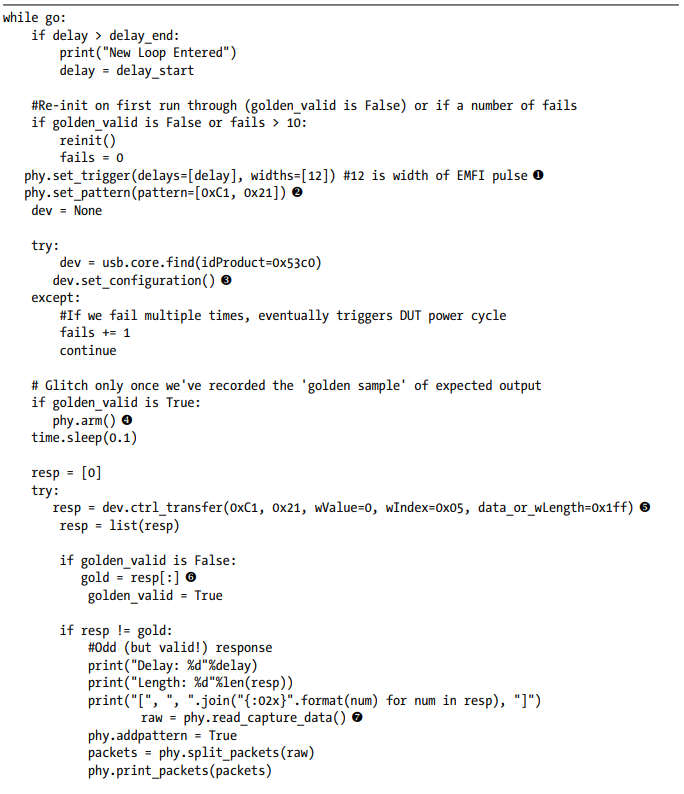
**O Código para Injeção de Falhas**

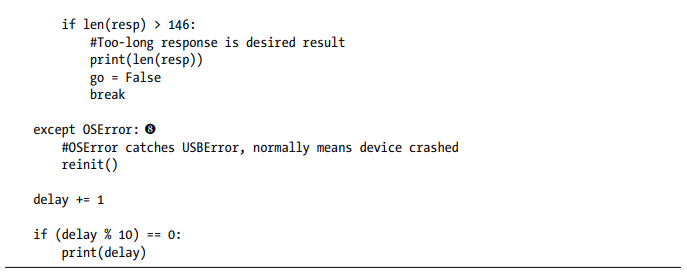
O script nos Listagens 7-11 e 7-12 (dividido para facilitar a leitura) nos permite reiniciar o dispositivo, emitir as solicitações WinUSB e acionar o ChipSHOUTER com base na solicitação WinUSB detectada no PhyWhisperer-USB.



Neste setup, utilizamos os recursos de controle de energia do dispositivo-alvo PhyWhisperer-USB, como evidenciado pela função reinit() que reinicia o dispositivo-alvo quando chamada. Essa função realiza a recuperação de erros quando o alvo trava. Um script mais robusto poderia reiniciar o dispositivo em cada tentativa, mas há um trade-off aqui, pois a reinicialização é a operação mais lenta no loop. Podemos tentar realizar um loop de glitch mais rápido apenas reiniciando o dispositivo quando o alvo parar de responder, mas o trade-off aqui é que não garantimos que o dispositivo esteja realmente iniciando no mesmo estado todas as vezes.

A Listagem 7-12 mostra o corpo real do loop do ataque.





A programação do tempo real da saída do gatilho em relação ao gatilho da mensagem USB e a largura do pulso EMFI são definidas. A largura (12) foi descoberta usando as técnicas discutidas anteriormente, principalmente ajustando a largura até vermos o dispositivo ser reiniciado (provavelmente um pulso muito largo!) e então reduzindo a largura até que o dispositivo pareça estar prestes a travar. Confirmamos que essa borda é uma largura bem-sucedida procurando por sinais de corrupção sem uma falha completa do dispositivo. Para o Trezor, podemos encontrar isso procurando por mensagens inválidas ou certas mensagens de erro sendo exibidas. Para ajustar a largura, não usamos o loop da Listagem 7-12. Em vez disso, inseriríamos o glitch durante a inicialização do dispositivo, quando ele estiver validando a memória interna. O Trezor exibe uma mensagem se a verificação de assinatura falhar, e poderíamos usar essa mensagem para indicar que encontramos bons parâmetros para nossa ferramenta EMFI que causarão uma falha neste dispositivo. A verificação de assinatura falhando na presença de um glitch provavelmente significa que de alguma forma afetamos o fluxo do programa (o suficiente para interromper a verificação de assinatura), mas o glitch não foi "muito forte" a ponto de causar uma falha no dispositivo.

O padrão de mensagem no qual nossa configuração está sendo acionada é definido, o que deve corresponder à posterior solicitação USB que estamos enviando para o dispositivo. Em cada iteração, o bootloader do Trezor é reconectado usando a chamada libusb dev.set\_configuration(), que também faz parte do tratamento de erros. Se esta linha lançar uma exceção, é provável que seja porque o stack USB do host não detectou o dispositivo.

Cuidado com o bloco except que suprime silenciosamente o erro logo após a chamada libusb. Esse bloco except assume que um ciclo de energia é suficiente para recuperar o alvo, mas se o stack USB do host travar, o script para de funcionar silenciosamente. Como mencionado antes, recomendamos executar isso em um sistema Unix bare-metal, pois o Windows geralmente causa problemas rapidamente devido ao stack USB do host bloquear o dispositivo após vários ciclos rápidos de desconexão/reconexão. Tivemos experiências igualmente negativas dentro de máquinas virtuais.

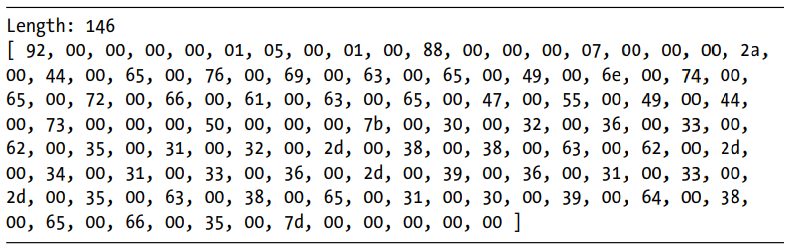
Para saber se o glitch teve algum efeito, mantemos uma "referência dourada" da resposta esperada da solicitação USB. O glitch real é inserido apenas quando a função arm() é chamada antes da solicitação USB. Na primeira vez, quando a referência dourada é obtida, a função arm() não é chamada para garantir que capturamos a saída não corrigida ("dourada").

Com essa referência dourada, agora podemos marcar qualquer resposta estranha. O tráfego USB que ocorreu durante a injeção de falha é impresso. Isso baixa os dados que foram automaticamente capturados quando a solicitação correspondeu ao padrão definido.

O código atualmente imprime informações apenas sobre respostas válidas. Você também pode querer imprimir capturas USB para respostas inválidas para determinar se a falha está inserindo erros. O PhyWhisperer-USB ainda captura os dados inválidos. Você precisaria mover a rotina de captura e impressão para o bloco de exceção OSError. Quaisquer erros irão ramificar o código para o bloco de exceção OSError, porque o stack USB não retorna dados parciais ou inválidos.

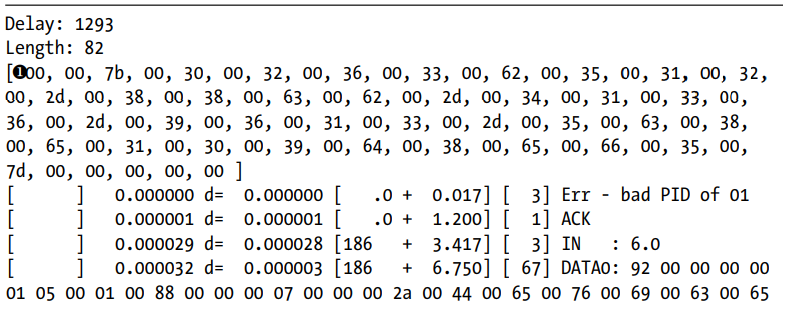
**Executando o Código**

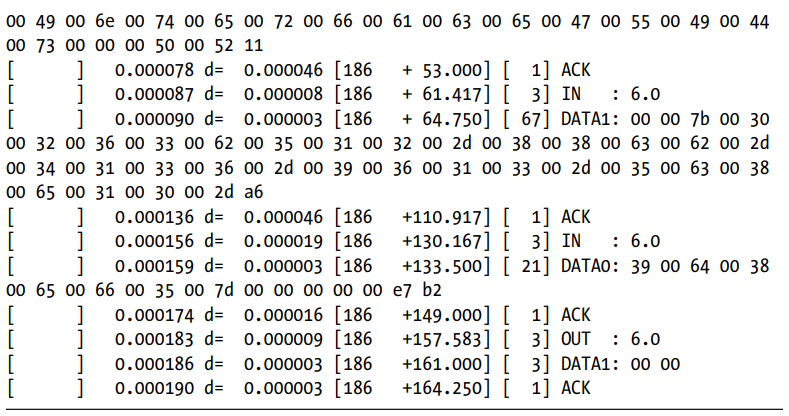
Como exemplo, a Listagem 7-13 mostra a referência dourada para a solicitação WinUSB.



Esta referência dourada é o valor dos dados retornados, então qualquer dado retornado que difira é esperado para indicar uma falha interessante (ou útil).

A Listagem 7-14 mostra uma condição repetível que observamos em um experimento. Os dados retornados (82 bytes) são mais curtos do que o comprimento da referência dourada (146 bytes).

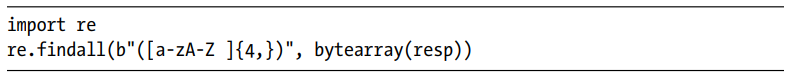




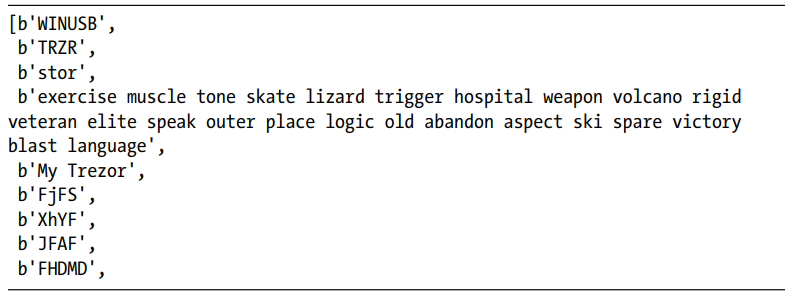
Os dados retornados são simplesmente a referência dourada sem os primeiros 64 bytes. Parece que uma transação USB IN inteira está faltando, o que sugere que uma transferência de dados USB inteira foi "pulada" nesta execução de injeção de falha. Como nenhum erro foi sinalizado nesta transferência, o dispositivo USB deve ter pensado que só deveria retornar o comprimento mais curto de dados. Tal falha é interessante, porque prova que as mudanças no fluxo do programa no dispositivo-alvo estão ocorrendo, o que é bom saber, pois mostra que nosso objetivo geral é razoável. Note novamente o erro de PID ruim, que se deve à falta da primeira parte de um pacote USB; está apenas no primeiro quadro decodificado e não é indicativo de um erro causado por uma falha.

**Confirmando um Despejo**

Como confirmamos que realmente tivemos um glitch bem-sucedido (e obtemos a semente de recuperação mágica)? Inicialmente, apenas procuramos por uma resposta "muito longa" e esperamos que a área de memória retornada inclua a semente de recuperação. Como a semente de recuperação secreta é armazenada como uma string legível por humanos, se tivéssemos um binário, simplesmente executaríamos strings na memória retornada. Como estamos implementando o ataque em Python, poderíamos usar o módulo re (expressão regular) em vez disso. Supondo que tenhamos uma lista de dados chamada resp (por exemplo, da Listagem 7-14), poderíamos simplesmente encontrar todas as strings com apenas letras ou espaços de comprimento quatro ou mais com uma expressão regular, como mostrado na Listagem 7-15.



Com sorte, obteremos uma lista de strings presentes nos dados retornados, como na Listagem 7-16.



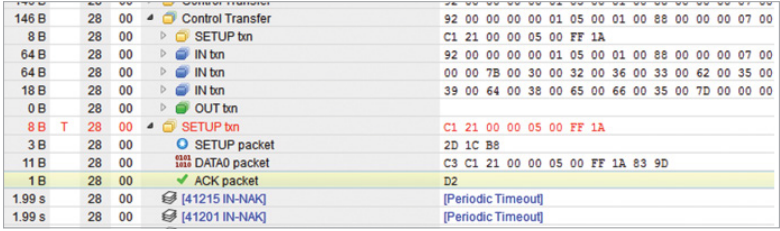
Uma das strings deve ser a semente de recuperação, que será a longa string de palavras em inglês. Ver isso significa um ataque bem-sucedido!

**Ajustando Finais o Pulso EM**

O último passo ao executar o experimento é ajustar finamente o próprio pulso EM, que neste caso significa digitalizar fisicamente a bobina acima da superfície, junto com ajustar a largura e o nível de potência do glitch. Podemos controlar a largura do glitch a partir do script PhyWhisperer-USB, mas o nível de potência é ajustado via interface serial ChipSHOUTER. Um glitch mais potente simplesmente provavelmente resetará o dispositivo, enquanto um glitch menos potente pode não ter efeito. Entre esses extremos, podemos ver indicações de que estamos injetando erros, como acionar manipuladores de erro ou causar respostas USB inválidas. Acionar manipuladores de erro indica que provavelmente não estamos reiniciando completamente o dispositivo, mas estamos tendo alguns efeitos sobre os dados internos sendo manipulados. No Trezor em particular, a tela LCD indica visualmente quando o dispositivo entrou em uma rotina de manipulador de erro e relata o tipo de erro. Novamente, o analisador de protocolo USB pode ser útil para ver se resultados inválidos ou estranhos estão ocorrendo. Encontrar um local que ocasionalmente entre em erro geralmente é um ponto de partida útil, pois isso sugere que a área é sensível, mas não é tão agressiva a ponto de causar falhas de memória ou de barramento 100 por cento do tempo.

**Ajustando o Timing com Base em Mensagens USB**

Um glitch bem-sucedido é aquele em que a solicitação USB passa com o comprimento total dos dados, tendo conseguido contornar a verificação de comprimento. Encontrar o timing exato requer alguma experimentação. Você terá muitos travamentos do sistema devido a erros de memória, falhas de hardware e reinicializações. Usando um analisador USB de hardware, você pode ver onde esses erros estão ocorrendo, o que ajuda a entender o timing do glitch, como mostrado anteriormente na Listagem 7-14. Sem o "truque" de poder modificar o código-fonte para descobrir o timing, seria absolutamente essencial entender onde esses erros estão ocorrendo; eles são indicadores que podemos usar para entender o timing.



**Sumário**

Neste capítulo, percorremos o processo de pegar uma carteira de bitcoin não modificada e encontrar a semente de recuperação armazenada nela. Utilizamos algumas características do design de código aberto do alvo para fornecer insights, embora o ataque pudesse ter sucesso sem essas informações. O design de código aberto do alvo significa que você também pode usá-lo como referência para investigar seus próprios produtos onde você tem acesso ao código-fonte. Em particular, mostramos como você poderia facilmente simular o efeito de uma injeção de falha usando um depurador conectado ao dispositivo.

Encontrar um timing de glitch bem-sucedido não é fácil. Os experimentos anteriores demonstraram quando a comparação estava ocorrendo, que é quando queremos que o glitch seja inserido. Como este tempo tinha jitter, não há um único tempo "correto". Além do tempo, é necessária alguma posição espacial. Se você tivesse uma mesa de digitalização XY controlada por computador, também poderia automatizar a busca pela localização correta. Neste exemplo, simplesmente usamos uma mesa manual, pois uma posição muito específica não pareceu necessária.

Novamente, devido à natureza do timing do glitch, tome cuidado para decidir sobre uma estratégia econômica de como procurar configurações de glitch candidatas. Você pode rapidamente ver que a combinação de localização física, tempo de glitch, largura de glitch e configurações de energia EMFI significa um grande número de parâmetros para pesquisar. Encontrar maneiras de estreitar o intervalo de pesquisa (como usar informações sobre estados de erro para entender zonas efetivas) é fundamental para manter o espaço do problema tratável. Registrar saídas "estranhas" também é útil ao investigar possíveis efeitos, porque se você estiver procurando apenas por uma faixa muito estreita de "sucesso", poderá perder alguns outros glitches úteis.

A taxa de sucesso final do despejo EMFI é baixa. Uma vez que os glitches tenham sido ajustados corretamente, 99,9 por cento dos glitches retornam um resultado que é muito curto e, portanto, não são bem-sucedidos. No entanto, podemos alcançar um glitch bem-sucedido em cerca de uma ou duas horas em média (após ajustar localização e timing), tornando-o um ataque relativamente útil na prática. Queremos destacar que, ao realizar injeção de falhas em dispositivos reais, uma parte significativa da engenharia reversa é realizada para descobrir o que pode ser falhado, como despejo USB, olhando para o código, e assim por diante. Esperamos que os capítulos anteriores tenham preparado você para alguns desses desafios, mas certamente encontrará desafios que não são cobertos aqui. Como sempre, tente reduzir os desafios à instância mais simples, resolva-os lá e, em seguida, mapeie-os de volta para o dispositivo completo.

Se você tentar recriar esse ataque completo, provavelmente o achará mais difícil do que os laboratórios que cobrimos no Capítulo 6, o que deve lhe dar uma ideia de como os ataques de falha em um dispositivo real podem ser mais difíceis na prática, mesmo que as operações fundamentais sejam semelhantes.

E agora por algo completamente diferente. No próximo capítulo, passaremos para a análise de canal lateral e mergulharemos em mais detalhes sobre o que aludimos nos capítulos anteriores: como a energia consumida por um dispositivo pode nos dizer tanto sobre as operações quanto sobre os dados sendo usados pelo dispositivo sob ataque.