Как устроен парсер:

Основная идея парсера в том, чтоб опознавать и пропускать сразу по несколько байт за раз. В случае http-запросов, при отсутсвии необходимости нормализовать строку мы можем сталкиватьсяс тремя ситуациями

1. нам необходимо распознать одну из множества строк, например GET, POST, HEAD
2. нам необходимо распознать последовательность символов, принадлежащих к определённому набору символов ASCII
3. нам необходимо пропустить пробелы.
4. нам необходимо как-то совмещать байты после скачков между буферами, если запрос разбит на несколько пакетов.

Для первого случая используется функционал, представленный в sse_parser.c функциями

int tokenSetLength(const char ** tokens) ;

TokenSet * initTokenSet(const char ** tokens, void * buf, int bufsize);

MatchResult matchTokenSet(const TokenSet * ts, Vector vec);

В парсере http_parser, повторяющем логику парсера tempesta, они редуцированы до наборов констант и вставок instrinsic-команд, т. к. набор сравниваемых строк незначителен.

Пусть набор строк для сравнения «GET, POST, HEAD, OPTIONS,PUT,»

Идея алгоритма сравнения такова:

Мы формируем маску способа расстановки байт для сравнения(M1), маску для группового сравнения(M2), и пару масок для сведения результата к ответу(M3,M4). Если текстовых констант много, то длина каждой маски будет больше 16 байт. За один проход сравнить все маски не получится, поэтому мы разбиваем маски на порции по 16 байт и последовательно обрабатываем их, сохраняя результаты от предыдущей порции в специальном регистре переноса.

Исходя из заданного набора строк, мы формируем 4 константы на каждую итерацию:

маску перестановки байт входной строки для последующего сравнения M1

	//GET,	//POST	//HEAD	//OPTI
итерация1:	0,1,2,0xFF,	0,1,2,3,	0,1,2,3,	0,1,2,3
	//ONS,	//PUT		
итерация2:	4,5,6,0xFF,	0,1,2,0xFF,	0xFFFFFFFF,	0xFFFFFFFF

маску ожидаемых значений байт после перестановки M2

итерация1:	'G','E','T',0,	'P','O','S','T',	'H','E','A','D',	'O','P','T','I'
итерация2:	'O','N','S',0,	'P','U','T',0,	0xFFFFFFFF,	0xFFFFFFFF

Далее мы сравниваем последнюю маску с перестановленными байтами группами по 4 байта

В случае строки POST результат будет выглядеть до сокращения так:

итерация1:	0x00000000,	0xFFFFFFFF,	0x00000000,	0x00000000
итерация2:	0x00000000,	0x00000000,	0x00000000,	0x00000000

А в случае строки OPTIONS результат будет выглядеть до сокращения так:

итерация1:	0x00000000,	0x00000000,	0x00000000,	0xFFFFFFFF
итерация2:	0xFFFFFFFF,	0x00000000,	0x00000000,	0x00000000

Далее мы сводим 4ки байт 0xFFFFFFFF->0xFFFF, 0x00000000->0x0000 используя результат сравнения от предыдущей и текущей итерации. Результат сравнения предыдущей операции хранится в переменной latch, туда же помещается результат сравнения от текущей итерации после сведения.

После этой операции для POST получаем:

итерация1: 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0xFFFF, 0x0000, 0x0000

Для OPTIONS:

итерация1: 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0xFFFF

итерация2: 0x0000, 0x0000, 0x0000, 0xFFFF,
0xFFFF, 0x0000, 0x0000, 0x0000

Следующим шагом «раскрашиваем» каждое поле с помощью вспомогательной маски

0x0001, 0x0002, 0x0004, 0x0008,
0x0010, 0x0020, 0x0040, 0x0080

И суммируем результаты операцией горизонтального сложения 16-битных слов hadd, одновременно размножая результат операции по всему регистру.

После этой операции для POST получаем:

итерация1: 0x0020, 0x0020, 0x0020, 0x0020,
0x0020, 0x0020, 0x0020, 0x0020

Для OPTIONS:

итерация1: 0x0000, 0x0010, 0x0010, 0x0010,
0x0010, 0x0010, 0x0010, 0x0010

итерация2: 0x0018, 0x0018, 0x0018, 0x0018,
0x0018, 0x0018, 0x0018, 0x0018

Далее мы выделяем результат. Для этого мы используем знание в какой позиции в маске M1/M2 стояла строка и к какому коду она приведёт. На основании знания этого кода, мы создаем маски M3/M4. В маске M3 будут содержаться ожидаемые биты, а в M4 — пара {длина строки, id строки}

M3-1: 0x0000, 0x0000, 0x0000, 0x0000,
0x0010, 0x0020, 0x0040, 0x0000,

M4-1: 0x0000, 0x0000, 0x0000, 0x0000,
0x0300, 0x0401, 0x0402, 0x0000,

M3-2: 0x0000, 0x0000, 0x0000, 0x0000,
0x0018, 0x0020, 0x0000, 0x0000,

M4-2: 0x0000, 0x0000, 0x0000, 0x0000,
0x0704, 0x0305, 0x0000, 0x0000,

Для POST уже на первой итерации сравнение с M3-1 даст нам ненулевые результаты, поэтому алгоритм завершится выдав ответ соответствующей позиции совпавшего слова, т. к. 0x0401: 4 байта, id = 1.

Для OPTIONS на второй итерации сравнение с M3-2 даст нам ненулевые результаты, поэтому алгоритм завершится выдав ответ соответствующей позиции совпавшего слова, т. к. 0x0704: 7 байт, id = 4.

Если ничего не найдено, алгоритм завершится, вернув 0. У алгоритма есть ограничение: строки длиной до 16 байт, плюс нельзя чтоб в списке входных строк одна строка была префиксом другой и шла впереди неё:

CANCEL, OK, OKAY, YES
но: CANCEL, OKAY, OK, YES

Для распознания последовательности символов, принадлежащих к определённому набору символов ASCII, используются функции

Vector matchSymbolsMask(SymbolMap sm, Vector v)

int matchSymbolsCount(SymbolMap sm, Vector v)

SymbolMap это переменная типа __m128i, содержащая битовую маску для 128 значений таблицы ASCII. Для того, чтоб понять как она организована, посмотрите на эту таблицу:

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Тип __m128i можно представить как unsigned char [16]. Каждому столбцу в вышеприведённой таблице соотвествует один байт, а каждой колонке — бит в байте. Почему так сделано? В SSE нет операции выбора бита из 128битного регистра, но есть возможность выбора байта. При этом используются только биты 0-3 для выбора номера байта, и бит 7 для вывода 0:

result[n] = b[n] & 0x80 ? 0x00: a[b[n][3:0]]

Мы используем бит 7 чтоб отсечь значения байтов 128-255, не попадающие в ASCII-таблицу. А биты 3-0 это как раз индекс столбца в таблице ASCII. Но дальше есть еще одна проблема: в SSE нет операции сдвига байт. Поэтому нам приходится эмулировать эту операцию с помощью сдвига входных байт на 4 вправо, и выборки байтов из маски

1,2,4,8,10,20,40,80,0,0,0,0,0,0,0,0

Объединяя результат операции AND над перестановленной маской и результатом выбора байтов из SymbolMap, мы получим набор нулевых и ненулевых байтов в позициях, соответствующих исходным байтам. Далее, мы можем получить значение типа int, путём сравнения результата с нулевым вектором и извлечения старших бит байтов результата. **Это будет инвертированная маска**. Если у вас выбрано меньше 16 байт, объедините результат с маской доступных байт:

result = _mm_movemask_epi8(result_mask);

result |= (0xFFFFFFFF<<bytes_available);

Из полученной маски можно получить количество совпавших байт командой __builtin_ctz.

Третья задача: пропуск пробелов – аналогична предыдущей, с той разницей, что мы можем сравнивать значения «в лоб» с вектором пробелов.

Четвертая задача состоит в том, чтоб сдвигать вектор байтов, досылая в него байты из другого вектора. В SSE есть необходимые команды, но они сдвигают только на постоянное колво байт. Можно конечно нагородить конструкцию на switch или if. Но мы используем константы:

0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,

0x03020100, 0x07060504, 0x0B0A0908, 0x0F0E0D0C,

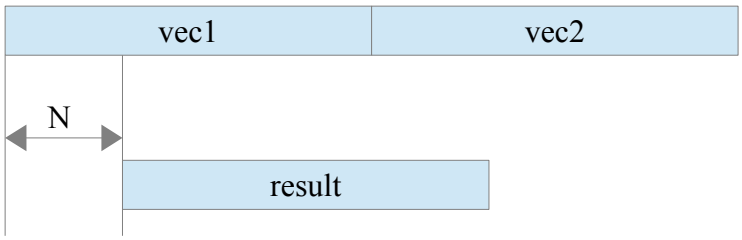
0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF

Если мы берем маску перестановки байт ровно с 16го байта константы, то она будет такой: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15. Это значит что каждый байт перейдет сам в себя и сдвиг будет равен 0.

Если мы берем маску перестановки байт с 17го байта константы, то она будет такой: 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0xFF. Это значит, что в 0й байт попадёт 1й байт, в 1й – 2й, и так далее, кроме 15го, куда запишут 0. Т.е. приращение смещения на 1 байт привело к сдвигу на 1 байт вправо. Если двигаться еще дальше(брать смещения 18,19,20 и т. д.) то регистр будет сдвинут еще больше.

Если мы берем маску перестановки байт с 14го байта константы, то она будет такой: 0xFF,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14. Это значит, что в 0й байт попадёт 0, в 1й байт – 0й байт, в 2й – 1й, и так далее. Т.е. уменьшение смещения на 1 байт привело к сдвигу на 1 байт влево.

Для совмещения двух векторов мы используем следующий приём:



vec1 = _mm_shuffle_epi8(vec1, _mm_right(N));
vec2 = _mm_shuffle_epi8(vec2, _mm_left(N));
result = _mm_or_si128(vec1, vec2);

где _mm_left и _mm_right это макросы для невыровненной загрузки маски сдвига со смещением 16+N и 16-N соответственно.

Теперь как мы из этого собираем парсер:

В начале парсера проверяем, не находимся ли мы в исходном состоянии:

```
    if (unlikely(state == Req_0)) {
```

Перед началом разбора запроса в FF-версии мы включаем логику быстрого проскакивания типовых запросов вида «GET http://».

```
#ifdef ENABLE_FAST_FORWARD
        if (len >= 16) {
```

Эта логика исходит из того, что большинство запросов к нам будут приходить достаточно большими пакетами(т.е. 16 байт мы наберем) и в них будет всего один пробел между методом а URL. Она быстро проскакивает сразу несколько состояний и выходит либо к разбору хоста либо к проверке наличи схемы(если пробелов всё таки больше 1). Как оказалось, эта логика заметно ускоряет прохождение тестов.

В начале цикла мы заранее подготавливаем указатель для создания fixup_pointeров Этот указазатель может быть за пределами текущего буфера: в дальнейшем код будет прибавлять к нему значения заведомо большие, нежели промах мимо начала буфера.

```
    for(;;) {
        unsigned char * fixup_ptr = data - bytes_cached;
```

Далее проверяем, не закончилась ли у нас еда для парсера, либо же парсер где-то решил что ему недостаточно байт. Например: имя метода GET приходит по байтам, и у нас сперва есть только буква G, затем буквы GE, затем GET, но еще нет пробела.

```
if (bytes_cached < 16) {
  if (unlikely(r == TFW_POSTPONE || (len + bytes_cached == 0)))
  {
```

В случае, если нам пора завершаться, мы fixupаем текущую строку. В tempesta как-то по особому fixupают заголовки, поэтому есть проверка — перед нами заголовок или просто часть запроса.

```
            if (parser->current_field) {
                if (parser->header_chunk_start) {
```

Если завершаться нам не пора, догрузим в регистр SSE еще байт:

```
        }
        int n = min(16 - bytes_cached, len);
```

Обычно мы просто читаем байты как есть, но в случае приближения к концу страницы мы можем получить page fault: для избежания этой ситуации последние 16 байт страницы считываются всегда в притык к её заднему краю и сдвигаются вправо, чтоб выглядеть так, как будто они были считаны как надо:

```
        //avoid page faults here
        long ldata = (long)data;
        __m128i compresult;
        if (unlikely(len < 16 && (ldata&0xFF0 > 0xFF0))) {
            compresult = _mm_lddqu_si128((const __m128i*)(ldata & ~0xFL));
            compresult = _mm_shuffle_epi8(compresult, _mm_right(ldata &
0xF));
        } else {
            compresult = _mm_lddqu_si128((const __m128i*)ldata);
        }
```

Собираем новый регистр из старых байт и новых:

```
        compresult = _mm_shuffle_epi8(compresult, _mm_left(bytes_cached));
        vec = _mm_or_si128(vec, compresult);
        bytes_cached += n;;
```

«Потребляем» входные данные:

```
        data += n;
        len -= n;
```

Может оказаться так, что состояние ожидающее определённую строку, получает байты которые его в теории устраивают, но конкретная комбинация не устраивает, а больше байт для этого состояния у нас нет: например вместо метода получили GETGETGETGETGETETGETGETGETGETGETGETGET

```
  if (unlikely(r == TFW_POSTPONE)) {
      r = TFW_BLOCK;
      break;
  }
```

Эта маска будет помогать нам поправлять результаты поиска символов с учетом того, сколько реальных байт у нас есть:

```
int avail_mask = 0xFFFFFFFF << bytes_cached;
```

Далее мы пропускаем пробелы если надо и пытаемся заново «подкачать» байты.

```
if (unlikely(state & Req_Spaces)) {
```

Если ни одного пробела не было, то и пропускать нечего:

```
    state &= ~ Req_Spaces;
}
```

Практически в любом состоянии нам нужно знать, сколько байт мы можем «потребить» и возможно, какое значение имеет байт или пара байт сразу после подходящих.

```
__m128i charset1 = __match_charset(_r_charset, vec, _r_cset1, _r_cset2);
int mask1 = (_mm_movemask_epi8(charset1))|avail_mask;
```

В FF-версии парсера, в состояниях, где парсер может без изменения состояния потребить сразу все 16 байт, есть код для быстрой «перемотки» входных данных.

```
if (state > Req_FastForward) {
    while (mask1 == 0xFFFF0000) {

        …

    }

    if (!bytes_cached) continue;
}
```

Превращаем маску в количество байт. Если байт набрано меньше 16, то нужно подсчитать, сколько именно байт нам подошло, и выделить первый неподошедший байт. Если выделять его позднее, это ведет к падению производительности.

```
int nchars1 = __builtin_ctz(mask1);

unsigned int lastchar = _mm_extract_epi16(
_mm_shuffle_epi8(vec, _mm_right(nchars1)), 0);
#define LAST ((unsigned char)lastchar)
#define LAST2 ((unsigned short)lastchar)
```

Далее идет finite-state-machine, которая предметно рассматривает каждое состояние. Она оперирует последними накопленными 16 байтами(в т.ч. байтами от предыдущих пакетов). У FSM меньше состояний, чем у «goto-парсера», т. к. она ожидаем увидеть на входе сразу достаточно байт для смены состояния. Если их недостаточно, то FSM сигнализует о необходимости докачать еще байт.

```
switch (state) {
```

Отдельно стоит отметить разбор схемы: в парсере темпесты есть ошибка, когда

название пути или хоста на букву h может быть воспринято как начало http://. Мы заранее считаем его именем хоста или пути пока не убедимся что перед нами именно http://.

Для разбора имени метода и версии HTTP мы используем приём номер 1, описанный выше.

Для разбора URI, Header name и Header value мы используем приём номер 3, описанный выше. Задача FSM на этом этапе просто понять что делать с неподходящим байтом.